

UNIVERSITY OF TARTU  
Faculty of Science and Technology  
Institute of Computer Science  
Software Engineering Curriculum

Manish Gupta

# Google Dataflow orchestration using TOSCA in the hybrid cloud

Master's Thesis (30 ECTS)

Supervisor(s): Chinmaya Dehury, PhD  
Pelle Jaakovits, PhD

Tartu 2022

## **Google Dataflow orchestration using TOSCA in the hybrid cloud**

### **Abstract:**

In today's world, data is as precious as oil. Many organizations depend on data to make critical business decisions, target specific customers, and accelerate their business growth. This importance of data leads to increased data creation and consumption volume. To process and provide logistics for this tremendous data, one requires a practical and automated approach to data handling. Data Pipeline is a series of interconnected modular tasks that collect, process and make data available to a wide array of systems with minimal manual intervention. There are numerous vendors and open-source platforms that support building Data Pipelines for an organization. However, developers need to have platform-specific knowledge to manage and orchestrate different data pipeline platforms. The lack of standardization for orchestrating data pipelines leads to increased development time and reduced reusability. TOSCA is an open standard used to define topology and orchestration specifications for different cloud services. In this paper, reusable TOSCA components were created in the RADON ecosystem to deploy, terminate, and manage Google Dataflow jobs. RADON is a research project that aims to develop a model-driven DevOps framework for serverless computing. The TOSCA components for Google Dataflow were designed to integrate with existing TOSCA components for Apache Nifi based data pipeline. The integration provides a one-stop solution for developers to build extensive data pipelines combining Google Dataflow and Apache Nifi.

### **Keywords:**

ETL, Data Pipeline, Google Dataflow, TOSCA, RADON

**CERCS:** P170 - Computer science, numerical analysis, systems, control

## **Google Dataflow orkestreerimine TOSCA abil hübriidpilves**

### **Lühikokkuvõte:**

Tänapäeva maailmas on andmed sama väärtuslikud kui nafta. Paljud organisatsioonid sõltuvad andmetest, et teha kriitilisi äriotsuseid, sihtida konkreetseid kliente ja kiirendada oma ärikasvu. See tähtsus suurendab andmete loomise ja tarbimise mahtu. Suurandmete töötlemine nõuab praktilist ja automatiseeritud lähenemist. Data Pipeline on rida omavahel ühendatud modulaarseid teenuseid, mis koguvad, töötlevad ja teevad minimaalse käsitsi sekkumisega andmeid kättesaadavaks paljudele süsteemidele. Paljud teenusepakujad ja avatud lähtekoodiga platformid pakuvad organisatsioonidele andmerude loomist. Arendajatel peavad aga olema platvormipõhised teadmised, et hallata ja korraldada erinevaid andmekanaleid. Puudub andmete torujuhtmete standard, mis pikendab arendust ning vähendab süsteemi taaskasutust. TOSCA on avatud standard, mida kasutatakse erinevate pilveteenuste topoloogia ja orkestratsiooni spetsifikatsioonide määratlemiseks. Selles artiklis loodi RADONi ökosüsteemis korduvkasutatavad TOSCA komponendid Google Dataflow tööde juurutamiseks, lõpetamiseks ja haldamiseks. RADON on uurimisprojekt, mille eesmärk on töötada välja mudelipõhine DevOps raamistik serverita andmetöötluse jaoks. Google Dataflow TOSCA komponendid loodi integreerimiseks Apache Nifi-põhise andmekanali olemasolevate TOSCA komponentidega, mis pakub arendajatele ühtset lahendust ulatuslike andmekanalite loomiseks.

### **Võtmesõnad:**

ETL, Data Pipeline, Google Dataflow, TOSCA, RADON

**CERCS:** P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	Problem statement . . . . .	6
1.2	Thesis contributions . . . . .	7
1.3	Thesis Outline . . . . .	8
<b>2</b>	<b>Background</b>	<b>9</b>
2.1	ETL . . . . .	9
2.2	Data pipeline . . . . .	10
2.2.1	Apache Nifi . . . . .	10
2.2.2	AWS Data Pipeline . . . . .	11
2.2.3	Google Cloud Composer . . . . .	12
2.2.4	Google Dataflow . . . . .	13
2.3	TOSCA . . . . .	13
<b>3</b>	<b>Related Work</b>	<b>17</b>
3.1	RADON Project . . . . .	17
3.2	RADON data pipeline Limitations . . . . .	19
<b>4</b>	<b>Methodology</b>	<b>20</b>
4.1	Existing TOSCA components in RADON data pipeline . . . . .	20
4.2	Developed TOSCA components for Google Dataflow . . . . .	22
4.2.1	Developed TOSCA node types for Google Dataflow . . . . .	22
4.2.2	Developed TOSCA relationship and capability types for Google Dataflow . . . . .	29
4.3	Interaction between Dataflow and developed TOSCA components . . . . .	32
4.4	Integrating Apache Nifi and Google Dataflow . . . . .	33
<b>5</b>	<b>Evaluation and Results</b>	<b>35</b>
5.1	Evaluation Setup . . . . .	36
5.2	Pub/Sub Subscription to BigQuery using Dataflow . . . . .	36
5.2.1	Prerequisites for this Dataflow . . . . .	38
5.2.2	Choosing the appropriate node types and supplying properties . . . . .	38
5.2.3	Designing the service template in the Winery . . . . .	42
5.2.4	Deploying the exported CSAR file using xOpera . . . . .	43
5.2.5	Evaluating final results . . . . .	44
5.3	Wordcount example using Dataflow (Batch) . . . . .	47
5.3.1	Prerequisites for this Dataflow . . . . .	48
5.3.2	Choosing the appropriate node types and supplying properties . . . . .	50
5.3.3	Designing the service template in the Winery . . . . .	53

5.3.4	Deploying the exported CSAR file using xOpera . . . . .	54
5.3.5	Evaluating final results . . . . .	55
5.4	Image Processing using Dataflow and Apache Nifi . . . . .	59
5.4.1	Prerequisites for this Dataflow . . . . .	60
5.4.2	Choosing the appropriate node types and supplying properties .	61
5.4.3	Designing the service template in the Winery . . . . .	67
5.4.4	Deploying the exported CSAR file using xOpera . . . . .	68
5.4.5	Evaluating final results . . . . .	68
<b>6</b>	<b>Conclusion and future work</b>	<b>73</b>
	<b>References</b>	<b>75</b>
	<b>Appendix</b>	<b>76</b>
I.	Glossary . . . . .	78
II.	Repositories . . . . .	78
III.	Technical Manual . . . . .	78
IV.	Licence . . . . .	83

# 1 Introduction

Digitization and recent data-driven approaches have led to an increase in data velocity and volume. Every organization that uses digital systems to assist in their daily operations generate and store essential data. This data is then processed to obtain valuable insights and make data-driven decisions. Organizations typically use different application systems for various departments or branches—most of the time, the data needed by various departments in an organization overlap. For example, consider a massive organization with multiple standalone systems that need to share data between them where the output of one system might be input to some other system. One must also build point-to-point integrations between systems to transfer the data. However, In today's world of Microservices architecture, this increases complexity due to the number of connections and data dependencies among systems. This dependency slows down the development effort, and the failure of one system might cause the whole architecture to fail. Additionally, expert data engineers must guarantee timely data collection and processing, make the data available to systems as required, and monitor data flow for errors, among other tasks, which is a lot of manual work.

Data Pipeline solves this problem of managing data in complex distributed systems. Data pipeline involves a set of storage and data processing activities connected in a sequential way to form a data flow that usually begins with a data source and ends in a data sink. The Data pipeline provides features like scheduled deployment and monitoring to ensure that the data-related tasks are performed reliably and with minimal human intervention. One can utilize cloud services to set up Data Pipeline to unlock cloud computing benefits like scalability, reliability, security, etc. Figure 1 shows an example of a Cloud Data pipeline where data is fetched from the Amazon S3 bucket, transformed, processed, and then the final results stored in a different Amazon S3 bucket.

Data pipelines may be set up over public cloud, private cloud, or a combination of both. A public cloud service is when cloud services are provided over the internet. In contrast, a private cloud service is the cloud infrastructure set up solely for an organization within a private network. Hybrid cloud encompasses a combination of both public and private clouds. Organizations may rely on a combination of private and numerous public cloud providers, i.e., hybrid cloud, for their data processing needs. Many organizations also rely on Cloud bursting, a technique of running the applications on private cloud and expanding into Public cloud as the demand spikes. Therefore, Data pipelines must support the hybrid cloud to support the all-inclusive data management in an organization.

## 1.1 Problem statement

Today, a typical organization has its data distributed across multiple cloud service providers and private clouds. Therefore, data pipelines may stretch over multiple cloud resources and data pipeline platforms. Studies have also shown that interoperability

between cloud platforms [1] remains one of the significant challenges for data pipelines. Moreover, data pipelines also have to deal with infrastructural challenges like difficulty integrating new sources, data pipeline scaling, and difficulty in adding new nodes[2]. These challenges call for a standard orchestration solution to deploy, terminate, and manage data pipelines across multiple cloud platforms.

Topology and Orchestration Specification for Cloud Applications (TOSCA) is an open standard that defines a cloud topology by dividing the cloud services into multiple reusable components. These reusable components interconnect together to form a flow of data and enable a fully automated deployment, orchestration, and termination of cloud services and data pipelines[3, 4, 5]. Various implementations of TOSCA such as SeaClouds[6] and ToscaMart[7] provide tools and resources to design complex cloud applications. RADON project[8, 9] proposes an architecture to design data pipelines on top of Apache Nifi and also allows deploying specific tasks in the AWS data pipeline.

However, there exists no TOSCA implementation for Google Dataflow, a proprietary data pipeline solution by Google. This limitation prohibits users from having a standard orchestration solution for Google Dataflow and may slow down the overall cloud application deployment process. A well-designed modular TOSCA implementation for Google Dataflow will allow users to build hybrid data pipelines that combine Dataflow and other data pipeline platforms for which TOSCA implementation exists. This thesis will develop TOSCA components for Google Dataflow by following a modular and extendible architecture. The development will allow users to design, develop, and maintain Google Dataflow faster and also make it possible to integrate Dataflow with existing TOSCA implementation for other data pipeline platforms.

## 1.2 Thesis contributions

This thesis will extend the RADON data pipeline architecture by adding Google Dataflow support to allow developers to build integrated data pipelines. Below we describe how this thesis contributes to the RADON project[8, 9] and makes data pipeline orchestration easier:

- (A) **Develop reusable TOSCA components for Google Dataflow** - This development will enable users to orchestrate their Google Dataflow for batch and streaming data using TOSCA. Users will have to supply python code or use Google-defined templates to run their Dataflow. The Dataflow node types will follow a modular and extendible architecture and thereby will be divided into Input, Processing, and Output block. In the case of Dataflow using python code, node types will convert the user-supplied Python code to the Dataflow template. The Dataflow template will then be executed using input and output from node types in Input and Output blocks. Users can also run Dataflow jobs using Google provided templates designed for basic operations like copying from one data source to another. Users

can supply the different data source and sink types using input and output node types.

- (B) **Enable integration for developed TOSCA components** - The developed Node Types for Google Dataflow will support integration with existing TOSCA components for Apache Nifi in the RADON ecosystem. For example, the input block from Google Dataflow can integrate with the output pipeline block from the Nifi data pipeline to receive data. Similarly, output block from Google Dataflow can integrate with input pipeline block from Nifi data pipeline to supply data. This integration will facilitate users to design data pipelines across Google Dataflow and Apache Nifi using a single interface.

### 1.3 Thesis Outline

This thesis is structured as follows. Section 2 provides an introduction to the ETL and data Pipeline concepts, discusses some state-of-the-art platforms and services that support building and managing data pipelines, and talks about a standard language to describe the topology of cloud-based services and their relationships. The section also expands on how the standard language makes it easy to orchestrate cloud services and build Data pipelines in the hybrid cloud. Section 3 describes the related works and tools that simplify building Data pipelines in hybrid cloud, the gap in that work, and how this thesis improves the existing work. Section 4 describes the developed tools that simplify building Data pipelines and how they can be modeled to integrate across multiple platforms. Section 5 describes the experimental setup and obtained results. Finally, section 6 discusses the conclusion and future work for this thesis.



## 2 Background

This section gives a brief overview of ETL and data pipeline concepts and also talks about some famous platforms/frameworks that allow managing Data pipelines. This section also describes how the TOSCA standard facilitates easy designing, automates pipeline deployment, and enables integration across multiple cloud platforms and frameworks.

### 2.1 ETL

ETL stands for Extraction, Transformation, and Loading of data. The ETL process begins with the extraction of data from one or more sources, where data extraction may be performed using the push or pull method. In the push method, the data source delivers the data to the ETL platform, whereas the ETL platform pulls the data in the pull method. After data extraction, the ETL platform transforms the raw data into a specified format as needed by applications and users. Finally, the Loading step in ETL loads the data to a target data warehouse.

The primary purpose of ETL may be to unify the data coming from multiple sources, transform it to a standard format, and make the data available to numerous users and applications so that meaningful information and statistics can be derived. An ETL pipeline may be scheduled to automate regular data operations with minimal human intervention. ETL operations can help to combine data from multiple sources into a common format and store it in a centralized data location such as a data warehouse. For example, a multi-branch bank may have its central data warehouse run daily ETL jobs to collect banking data from different branches, transform it to a common format, save the transformed data, and generate consolidated reports.

An ETL job is usually done in batches meaning that the data is extracted in one chunk at a time. However, ETL pipelines do not support streaming data, such as continuous data coming from an IoT device. Moreover, ETL jobs are restricted to performing only a specific sequence of tasks that must end by storing the data into a target system. Therefore, ETL pipelines are not suitable to manage data in complex event-driven systems. ETL jobs are also not suitable in complex distributed data systems where the data should be processed multiple times and made available to different systems. To overcome these limitations, users may design data pipelines that support both streaming and batch data. A data pipeline may be considered a more advanced version of ETL that can be configured to move data across multiple destinations along with multiple data processing steps and transformations in between.

## 2.2 Data pipeline

Brik et al. [10] describes Data pipelines as software systems that process collections of data from multiple sources to produce either transformed data, aggregate data, or resulting data from applied functions. The data processing may consist of hundreds of jobs performed sequentially. Data pipelines can handle both batch data and streaming data. Organizations constantly generate data and process them to derive valuable meanings. Data handling complexity grows exponentially as the data size and number of data sources increase. Data pipelines are a streamlined way of managing data in an organization. It provides features like easy deployment, reduced complexity, end-to-end data monitoring, scheduling jobs, and improved data traceability. Figure 1 shows the logical architecture of a cloud-based sample Data pipeline where data is fetched from the Amazon S3 bucket, transformed, processed. Then the final results are stored in a different Amazon S3 bucket.

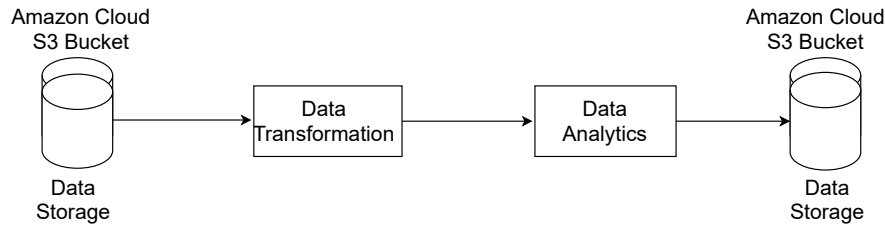


Figure 1. A typical Cloud Data Pipeline

Numerous open-source and proprietary data pipeline platforms support designing data pipelines. Some open-source platforms and technologies that support building data pipelines are Apache Nifi and Apache Beam, whereas AWS data pipeline and Google Dataflow are vendor-specific data pipeline technologies. Below some major data pipeline technologies are discussed.

### 2.2.1 Apache Nifi

Apache Nifi [11] is an open-source platform for managing the flow of data across several systems. It also provides a web interface to facilitate building data pipelines as a connected network of Processors. Apache Nifi represents the data flowing through the pipeline as Flowfile, consisting of a pointer to the data and dynamic attributes like data priority and data size. Processors in Apache Nifi are designed to perform specific tasks on data (or Flowfiles). For example, a *putFTP* processor sends data to an FTP server, and a *fetchFTP* can receive the data from an FTP server. Apache Nifi consists of several built-in processors which can be combined to build extensive data pipelines. Processors are linked using Connection which acts as a queue and manages the transfer rate of Flowfiles between processors. Apache Nifi also supports Process Group, a collection of

Processors and Connections attached to form a Data pipeline. Process group acts as a BlackBox that receives input data and emits output data. The created process group can be reused multiple times in the data pipeline to create even bigger data pipelines. Apache Nifi also enables users to export their Data pipeline design as an XML file. This feature enables data pipeline authors to share their designs across teams and promote reusability.

Figure 2 demonstrates a screenshot for Apache Nifi web interface containing a simple data pipeline. The data pipeline consists of a *GenerateFlowFile* processor on the left which has a directed connection towards the *PutS3Object* processor. In this data pipeline, *GenerateFlowFile* processor generates random Flowfiles and then pushes the Flowfile to the *PutS3Object* processor via the relationship named as success in the given figure. The *PutS3Object* processor then publishes the generated Flowfiles to the AWS S3 bucket.

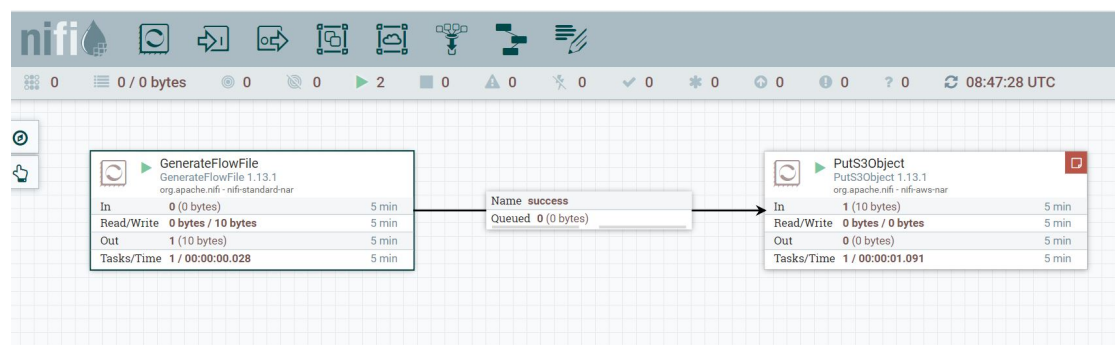


Figure 2. A demonstration of Apache Nifi data pipeline

### 2.2.2 AWS Data Pipeline

AWS Data pipeline [12] is a proprietary data pipeline solution by Amazon cloud services that allows building data pipelines across AWS cloud resources. One can also use pre-defined templates to design simple data pipelines or use the architect to design complex data pipelines as per their need. Users can configure their data pipelines to run immediately or schedule them to run at specific times or based on any event. The two main components of the AWS Data pipeline are:

(A) **Data Nodes** - The location in the Amazon cloud where the data is stored. Available options for Data Nodes are:

- *DynamoDBDataNode* - A Dynamo DB table.
- *SqlDataNode* - An SQL table.
- *RedshiftDataNode* - An Amazon Redshift table.

- ***S3DataNode*** - An Amazon S3 location.

(B) **Activities** - Activities are the specific tasks that are performed on data stored in Data Nodes. AWS data pipeline Activity uses computation resources, namely 'resource,' to perform the assigned task. Supported resource types in the AWS data pipeline are EC2 instance and EMR cluster.

Supported AWS Data pipeline activities are:

- ***CopyActivity*** - Copies data from one Data Node to another.
- ***EmrActivity*** - Runs an Amazon EMR cluster.
- ***HiveActivity*** - Executes a Hive query on an Amazon EMR cluster.
- ***HiveCopyActivity*** - Executes a Hive query on an Amazon EMR cluster and supports advanced data filtering.
- ***PigActivity*** - Executes a Pig script on an Amazon EMR cluster.
- ***RedShiftCopyActivity*** - Copy data from one Amazon Reshift table to another.
- ***ShellCommandActivity*** - Executes a custom shell command as an activity.
- ***SqlActivity*** - Executes a SQL query on a database.

Users can write a JSON file in the specified format to design customized data pipelines. AWS data pipeline provides a graphical interface to simplify creating the Directed Acyclic Graph (DAG) for data pipeline jobs. AWS data pipeline also supports scheduling for created data pipelines to execute data pipelines automatically at specific times or based on some events. Moreover, AWS data pipeline provisions the cloud resources and infrastructure on the cloud, meaning the data pipeline jobs are highly scalable. However, AWS data pipeline is a proprietary solution from Amazon, and it is challenging to use AWS data pipeline in conjunction with cloud services from other providers.

### 2.2.3 Google Cloud Composer

Cloud Composer [13] is a Google service that is an implementation of the open-source project Apache Airflow [14] running on top of Google Cloud resources. It is a fully managed workflow orchestrator that helps create, schedule, and monitor data pipelines and complex workflows across multiple cloud vendors. Cloud Composer is similar to Apache Airflow, except in Cloud Composer, Google manages the resource provisioning and platform setup. So, users can focus on building workflows via the Airflow web interface or command-line tools.

In Cloud composer, workflows are a series of tasks represented using Directed Acyclic Graphs, or DAGs. Tasks could involve ingesting, transforming, analyzing, or publishing

data, among many other options. Workflows can also be designed to create data pipelines. Cloud Composer's user-friendly interface makes it easier to deploy workflows with just a drag-and-drop feature or python program. It also makes it easier to set up the workflow environment and add required python libraries on the go. Cloud Composer provisions Google cloud resources like Bigquery, Dataflow, Cloud Storage, Datastore, Pub/Sub, Dataproc, and Cloud ML Engine to run workflows. So, it is not easy to estimate the cost of running workflows beforehand.

#### **2.2.4 Google Dataflow**

Google Dataflow [15] is a serverless data processing platform that can process both batch and stream data. Users need to write data processing jobs using Apache Beam [footnote Apache beam SDK] library in Python or Java. The written code describes a sequence of tasks and runtime parameters for the desired job in the Dataflow. Google Dataflow then converts the written code into a Dataflow template. After that, the template can be executed to create data processing jobs. Users may write the python code so that the Dataflow job execution will require runtime parameters. In such cases, users must provide runtime parameters to trigger the Dataflow job execution. Usually, the runtime parameters are the input and output location. Google also provides a set of pre-defined templates for some generic tasks like copying the data from one location and data type to other. These Google-provided templates are beneficial for users from a non-programming background. Google Dataflow automatically provisions cloud resources and manages clusters for running the specified job. Dataflow automatically upscales or downscales the number of worker instances executing the job based on the data traffic.

One of the drawbacks of using Google Dataflow is that user needs to have programmatic knowledge to design data pipelines using the Apache Beam library. Moreover, unlike the AWS data pipeline, Google Dataflow does not support inbuilt scheduling services for the Dataflow job. Therefore, users need to find other ways to schedule their Dataflow jobs. Some ways to schedule Dataflow jobs include using a Cloud scheduler like terraform or running a cron job process.

### **2.3 TOSCA**

Industrial data pipelines are typically complex and may use multiple distributed applications connected sequentially to design workflows. Developers need to deploy the individual applications first in a specific sequence before deploying data pipeline components. Each of these application may have different constraints; for example, A MySQL database may need a computing instance with higher computing power and additional connected storage. This complexity makes it very difficult to deploy these applications manually one by one in the correct order. The overall deployment process is error-prone. As the deployment is platform-specific, engineers have to start from scratch to deploy

a similar setup in a different cloud platform. Orchestration is the process of automatic configuration, management, and deployment of related applications and services. The complexity in industrial application deployment shows the need for an orchestration specification that can be reused across multiple cloud service providers.

TOSCA, which stands for Topology and Orchestration Specification for Cloud Applications, is an open standard to describe the application topology in the cloud by dividing the applications and infrastructural resources into multiple components. Each component has its properties, attributes, requirements, capabilities, connections, and dependencies. Therefore, engineers can specify their application topology using TOSCA, which can then be deployed across multiple cloud platforms in an automated fashion.

The TOSCA metamodel [3] describes the application topology using Service Templates. Service templates include node types that define a component in an application topology, and it also includes relationship types that describe the relationship between node types. Both node types and relationship types may define lifecycle operations using scripts that a TOSCA compliant orchestration engine may invoke during the instantiation of the service template.

For example, Listing 1 illustrates a Service Template to instantiate a Java application. The nodes and their relationships as shown in the service template are:

- *Server* node to instantiate a computing physical server.
- *Apache* node to instantiate an Apache web server and host it on a *Server* node.
- *MyJavaApplication* node to instantiate the Java application and host it on top of *Apache* node.

Node types may also define lifecycle operations like create, configure, start, stop, or delete. For example, *MyJavaApplication* node type defines a create method to install the Java application, configure method to configure the java application application based on user requirements, and finally start method to start the Java applications.

---

```
tosca_definitions_version: tosca_simple_yaml_1_3
description: Serive Template for deploying MyJavaApplication.
topology_template:
  inputs:
    application_admin_username:
      type: string
    application_admin_password:
      type: string
    #omitted for brevity
  node_templates:
    Server:
      type: tosca.nodes.Compute
      #omitted for brevity
```

```

Apache:
  type: tosca.nodes.WebServer.Apache
  requirements:
    - host: Server
  #omitted for brevity
MyJavaApplication:
  type: custom.nodes.JavaApplication
  derived_from: tosca.nodes.Root
  properties:
    application_admin_username: { get_input:
      application_admin_username }
    application_admin_password: { get_input:
      application_admin_password }
  requirements:
    - host: apache
  interfaces:
    Standard:
      inputs:
        username: { get_property: [ SELF,
          application_admin_username ] }
        password: { get_property: [ SELF,
          application_admin_password ] }
      operations:
        create:
          implementation:
            primary: create
        configure:
          implementation:
            primary: configure
        start:
          implementation:
            primary: start
  #omitted for brevity

```

---

Listing 1. A sample TOSCA Service Template

A node type may specify a requirement on a node or multiple nodes. These requirements are satisfied using different relationship types, including Hosted on, Depends on, Connects to, Attaches to, Route to, and any other custom relationship. The requirements for a node type can only be satisfied by a node type with an equivalent capability type. Relationship in the TOSCA decides the order of execution for node types. In the Listing 1, *Apache* node type is connected to *Server* node type using the relationship type *HostedOn* which means *Apache* node type is hosted on top of the *Server* node type. Therefore, the orchestrator instantiates *Server* node type before *Apache* node type. A similar relationship applies between *MyJavaApplication* node type and *Apache* node type, as Java application is hosted on the Apache web server.

Graphical Modeling Tools like Eclipse Winery allows users to assist users in visually

designing complex cloud applications using TOSCA components. Users combine node types and relationship types to create the service template reflecting their application topology. Finally, the lifecycle operation scripts and additional artifacts for application deployment are packaged together with the Service template to create a TOSCA archive called Cloud Service Archive (CSAR) file. The TOSCA-compliant orchestration engine can then deploy the CSAR file to create and manage cloud applications. Figure 3 illustrates the Winery web interface.

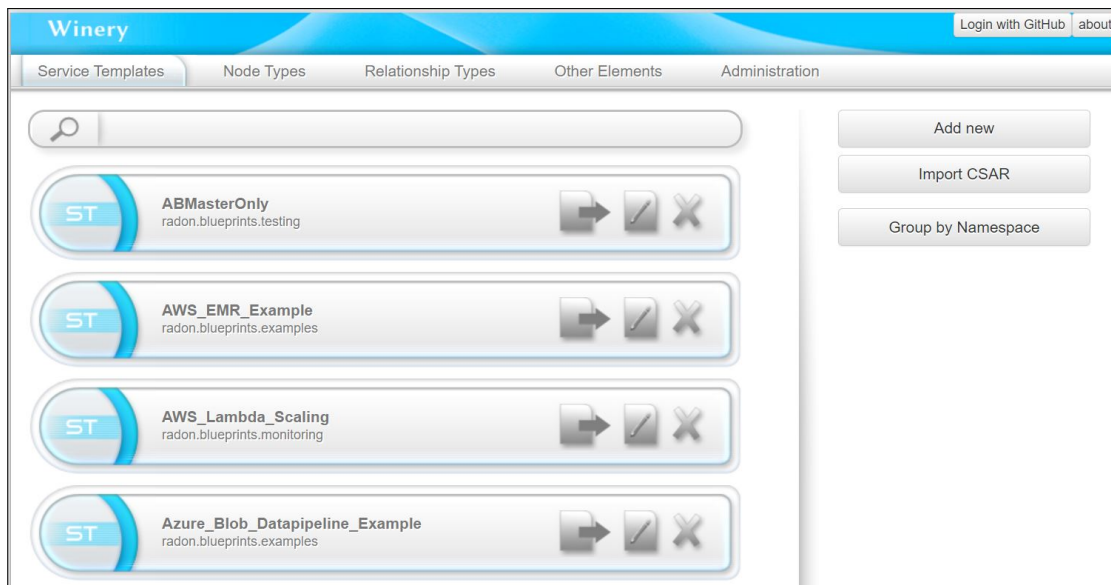


Figure 3. Winery web interface

### Alternatives to TOSCA

TOSCA is by far the most widely used orchestration standard. However, other orchestration standards exist, such as OpenStack heat <sup>1</sup> and Amazon AWS CloudFormation <sup>2</sup>. These orchestration standards are specific to their platform and do not support other cloud providers. OpenStack heat does provide compatibility with the AWS CloudFormation, but it mainly provides services for applications within the OpenStack cloud. Similarly, AWS CloudFormation is mainly designed for AWS cloud services. Therefore, this thesis considers TOSCA to build data pipelines in the hybrid cloud as TOSCA is compatible across multiple cloud providers.

<sup>1</sup><https://wiki.openstack.org/wiki/Heat>

<sup>2</sup><https://docs.aws.amazon.com/AWSCloudFormation/latest/APIReference/Welcome.html>



### 3 Related Work

Multiple industrial and academic projects aim to utilize TOSCA to provide an orchestration framework for building portable and complex cloud applications. SeaClouds[6] provides an open-source framework to deploy and manage cloud applications over multiple cloud providers. SeaClouds allows developers to integrate IaaS, PaaS, and SaaS from different cloud service providers under a single interface making the cloud deployment process portable. SeaCloud uses TOSCA under the hood to define application component specifications and their interdependencies. Similarly, TOSCAMART[7] is a method that allows developers to design complex cloud applications by reusing existing TOSCA specifications for application components. Although much research has been done to simplify deploying cloud applications in general, few studies specifically focus on the automatic orchestration of data pipelines. RADON is one such research project that focuses on simplifying building data pipelines using the TOSCA standard. This section will discuss the RADON project and its framework for orchestrating data pipelines.

#### 3.1 RADON Project

RADON<sup>3</sup> research project is working on making the benefits of serverless computing available for the European software industry. Casale et al. [16] introduce the RADON project as research work that aims to develop a model-driven DevOps framework for serverless computing. The goal of the RADON project is to allow developers to deploy cloud applications rapidly using RADON-developed independent microservices. The project hopes to solve the complexity of building serverless cloud applications by providing complete DevOps frameworks that include the modeling environment, the runtime environment, and the coding environment. The environments will facilitate the development and release of FaaS-based applications.

Dehury et al. [8] presents a RADON data pipeline architecture using the TOSCA standard to utilize serverless platforms. The proposed architecture uses Apache Nifi under the hood to automate the ingestion, transformation, and routing of data between multiple platforms and services. The RADON project also provides reusable TOSCA definitions and blueprints required to design the proposed data pipelines. RADON provides numerous reusable TOSCA components (node types and relationship types) that integrate together to create complex data pipelines using Apache Nifi under the hood. RADON also provides support for AWS Data pipeline, using which users can automate the orchestration of AWS data pipelines. However, the AWS data pipeline built through RADON is inflexible and can be used to perform a specific task.

RADON project also supports numerous tools that help users to build cloud applications easily. Some RADON-supported tools that benefit users to design and maintain

---

<sup>3</sup><https://radon-h2020.eu/>

cloud applications in the RADON ecosystem are described below.

- (A) **Graphical Modeling Tool** - RADON supports Eclipse Winery, which is a graphical modeling tool. It provides an easy web interface to build complex cloud application architecture using TOSCA node types and relationship types. It also allows the creation of new node types and relationship types. Eclipse Winery also allows importing and exporting CSAR files for service templates, node types, and relationship types. This thesis will use Eclipse Winery to develop node types, relationship types, and service templates to develop data pipelines.
- (B) **RADON orchestrator** - xOpera [17] is an open-source and lightweight RADON orchestrator that is compliant with the TOSCA specification. xOpera supports TOSCA Simple Profile in YAML Version 1.3 [3]. xOpera supports Ansible automation to implement the TOSCA standard. Many other TOSCA-compliant orchestrators exist, like Ystia Orchestrator (Yorc), Cloudify, and Ubicity. However, this thesis will use the xOpera orchestrator for its experiments as xOpera is lightweight, quick to install, and easy to use.
- (C) **Template library** - RADON provides a Github repository <sup>4</sup> that contains reusable TOSCA components. The reusable TOSCA blueprints are modular, and they can combine to develop complex cloud applications. RADON provides support to build data pipelines and cloud infrastructure over different vendors. Developers can either use these templates to speed up their cloud orchestration process. Developers can also design their custom TOSCA components building on top of existing templates.

### Why RADON

With so many cloud applications and frameworks popping up every year, industries need a one-stop solution for designing and maintaining complex cloud applications. RADON provides numerous tools and reusable cloud components that make building cloud applications across multiple cloud platforms easier. RADON has already implemented a data pipeline architecture that utilizes the Apache Nifi platform to create complex data pipe-lines. However, some gaps need to be filled to create comprehensive data pipelines incorporating different cloud platforms. This thesis will add features to the RADON project because there is a need for a one-stop cloud orchestration solution for hybrid cloud in industries. The RADON data pipeline limitations and contributions are discussed in the subsequent section.

---

<sup>4</sup><https://github.com/radon-h2020/radon-particles>

## 3.2 RADON data pipeline Limitations

Google Dataflow is one of the widely used services for building data pipelines in industries. Small and Medium Enterprises (SME) prefer these services as they remove the overhead of provisioning resources for running data pipelines. Some organizations also use Google Dataflow as a part of one big data pipeline, which spreads over many other cloud platforms. However, RADON does not provide Node Types and Relationship Types that facilitate building data pipelines in Google Dataflow.

As discussed, RADON does enable users to design Apache Nifi based data pipelines that utilize the power of serverless platforms. However, this limitation restricts users from designing data pipelines that expand over multiple platforms, stretching from Nifi based data pipelines to Google Dataflow. Moreover, the RADON particle <sup>5</sup> does contain Node Types for deploying AWS data pipelines. However, Node Types built for AWS data pipeline are standalone, meaning all the created Node Types perform a very specific task. These created Node Types do not integrate with other Node Types to facilitate designing data pipelines across multiple platforms. The standalone Node Types are against the TOSCA principal, promoting modularity where each Node Type can connect with other Node Types to form a complex cloud application.

---

<sup>5</sup><https://github.com/radon-h2020/radon-particles>

## 4 Methodology

This section describes the existing RADON data pipeline architecture and also discusses how this thesis adds to the current architecture by developing reusable TOSCA components to orchestrate Google Dataflow. The section describes the developed node types and relationship types in detail. It also provides high-level architecture on how developed TOSCA components for Dataflow automate the Google Dataflow orchestration under the hood. It also discusses how the developed TOSCA components for Google Dataflow can integrate with existing RADON data pipeline components to create hybrid data pipelines.

### 4.1 Existing TOSCA components in RADON data pipeline

Dehury et al. [18, 9] describes the data pipeline architecture as a combination of *PipelineBlocks*, *InputPipe*, and *OutputPipe*. *InputPipe* serves as the gateway to ingest data into the *PipelineBlock*. Similarly, *OutputPipe* pushes the data out of the *PipelineBlock*. The *PipelineBlock* is the crucial processing hub for the data pipeline, a series of interconnected tasks implemented on top of Apache Nifi. Each component in *PipelineBlock* is equivalent to a task in Apache Nifi, or an activity in the AWS data pipeline. The uncolored components in Figure 4 represent the node types hierarchy for the existing RADON *PipelineBlock*. The existing *PipelineBlock* can be categorized into *SourcePB*, *MidwayPB*, and *DestinationPB*, which are described below.

- (A) ***SourcePB*** - *SourcePB* acts as the starting point for *PipelineBlock*, which denotes reading the data from a data source. *SourcePB* does not receive any incoming connection as it is considered the data source. However, it may support outgoing connections to either *MidwayPB* or *DestinationPB*. This node type is further categorized into *ConsumeRemote* and *ConsumeLocal*. As the name suggests, *ConsumeLocal* gets the input from a local network while *ConsumeRemote* fetches the input from a remote source such as Google Cloud Storage or AWS S3. This node type defines the input location and other details required to fetch input as its TOSCA property.
- (B) ***MidwayPB*** - This *PipelineBlock* represents data processing tasks. Data processing may happen locally or using a remote function such as FaaS. Therefore, *MidwayPB* is divided into *LocalAction* and *RemoteAction*. There is also a third type of *MidwayPB*, namely *RouteToRemote*, responsible for routing the data from a local host to a remote host. *RemoteAction* represents invoking a FaaS function such as the AWS Lambda function or OpenFaaS function. A *MidwayPB* has an incoming connection from *SourcePB* and one or more outgoing connections to *DestinationPB*.

(C) **DestinationPB** - The *DestinationPB* is the final block in the data pipeline, representing publishing data to a local or remote endpoint. This node type is categorized into *PublishRemote* and *PublishLocal* based on their functionality. Users may provide the output location and other required parameters to publish data as the TOSCA property for the appropriate node type.

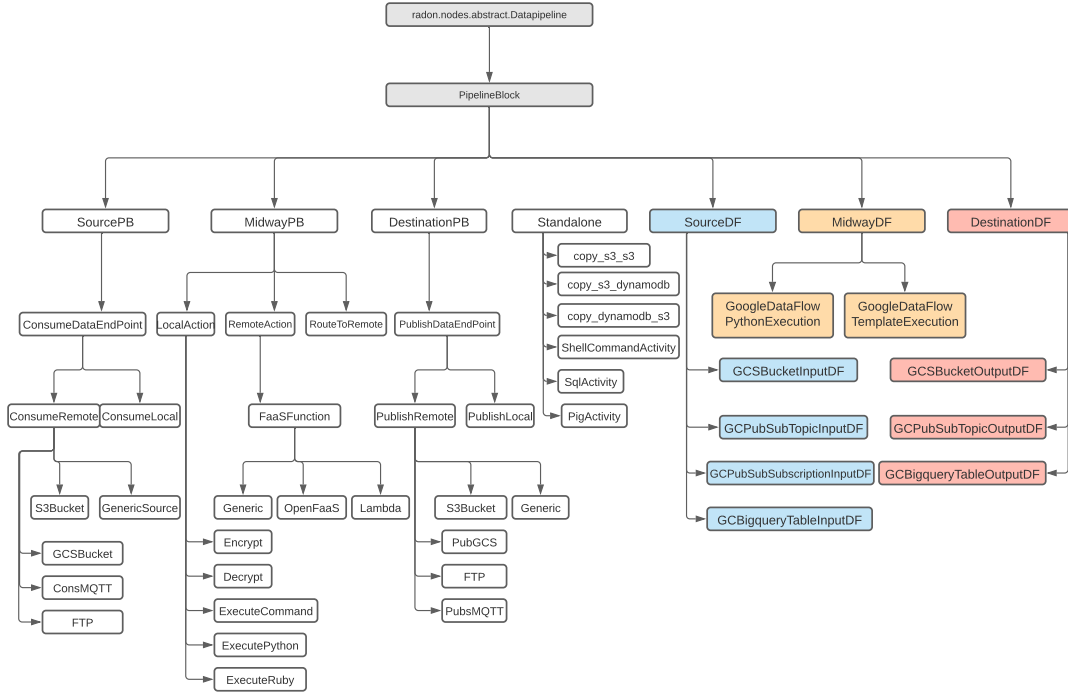


Figure 4. TOSCA-based pipeline models hierarchy for RADON

### Other existing TOSCA components in RADON ecosystem

RADON also provides some vendor-specific TOSCA templates that may assist further in developing data pipelines. RADON supports orchestrating cloud resources for AWS, Google Cloud, and Microsoft Azure. It contains TOSCA components to deploy and configure cloud resources such as Google Cloud Storage, AWS S3 Bucket, Google Cloud Function, AWS Lambda, etc. These TOSCA components may be used in conjunction with the RADON data pipeline to make development easier. For example, user may set up a GCS Bucket before deploying a data pipeline that writes to the created GCS bucket.

## 4.2 Developed TOSCA components for Google Dataflow

This thesis creates reusable TOSCA node types, relationship types, and capability types to allow users to design Google Dataflow. The developed TOSCA components are described below in detail.

### 4.2.1 Developed TOSCA node types for Google Dataflow

Developed node types for Google Dataflow follow a similar approach as the existing RADON data pipeline to categorize the developed node types based on their purpose. Figure 4 shows the categorization of newly developed node types as colored blocks and also shows the existing node types for the RADON data pipeline. The newly developed node types derive from *PipelineBlock* node type as part of the RADON data pipeline. The newly developed TOSCA node types are categorized into *SourceDF*, *MidwayDF*, and *DestinationDF*.

#### ***SourceDF***

*SourceDF* is an abstract parent node type for all the node types that collect user input parameters for the Google Dataflow job. The *sourceDF* node type requires an outgoing connection to the *MidwayDF* node type. This connection is of many-to-many nature, meaning that the same *SourceDF* node type may have outgoing connections to multiple *MidwayDF* node types, and a *MidwayDF* node type can receive income connections from multiple *SourceDF*. *SourceDF* contains four types of node type implementations, namely *GCSBucketInputDF*, *GCPubSubTopicInputDF*, *GCPubSubSubscriptionInputDF*, and *GCBigqueryTableInputDF*. These node types collect input from source types, including Google Cloud Storage, Google Cloud Pub/Sub, and Google Cloud Bigquery table. The *SourceDF* node types do not implement any life cycle operations, and it simply provides the input parameters to the *MidwayDF* via its properties. The available *SourceDF* node types are discussed below.

- (A) ***GCSBucketInputDF*** - This node type is used to supply input GCS bucket path for the Dataflow. This node also receives the parameter name described as runtime argument in Dataflow Python code or Dataflow template.
- (B) ***GCPubSubTopicInputDF*** – This node type receives input Topic that receives the input streaming data for the Dataflow. It also receives the parameter name, which is described as a runtime input argument in Dataflow Python code or Dataflow template.
- (C) ***GCPubSubSubscriptionInputDF*** - Similar to *GCPubSubTopicInputDF* node type, this node type receives input Subscription for the Dataflow.

- (D) ***GCBigqueryTableInputDF*** – This node type is used to receive input Bigquery table for the Dataflow.

### ***MidwayDF***

This *PipelineBlock* is at the center of the Dataflow processing, and it is responsible for the Python to Template conversion and triggering the Dataflow execution using the generated template. This node type receives the input parameters from *SourceDF* and output parameters from *DestinationDF*. It has a many-to-many relation with *SourceDF* and *DestinationDF*. It means that this *PipelineBlock* can receive multiple input sources and multiple output destinations. This node type receives the path of the Google credential file, enabling the TOSCA lifecycle operations to make authenticated interactions with the Google cloud services. The *MidwayDF* has two concrete implementations, which are briefly described below.

- (A) ***GoogleDataFlowPythonExecution*** - This node type receives the absolute path of the Python code. If multi-file Python code is designed for the Dataflow, then the absolute path for the entry file should be supplied. Developers also need to include a setup file including the required Python dependencies for the Dataflow job. The Python code is converted to a template file and stored at a user-specified Google cloud storage using this node type's life cycle operation. The node type then triggers the Dataflow job using the converted template file and other required parameters. Users should use this node type for designing data pipelines using Python code. Below we discuss the Standard life cycle operations created for this node type.

- **Create** - The create life-cycle operation is implemented using Ansible script and is responsible for creating the template out of supplied Python code. This operation uses Python3 to install all the Python dependencies that are supplied as the property. The script configures the `GOOGLE_APPLICATION_CREDENTIALS` environment variable to the path of the Google credentials file and executes the Python code to generate a Dataflow template. Listing 2 shows the ansible script to implement create operation.
- **Configure** - The configure life-cycle operation is implemented using Ansible script and is responsible for configuring the Google Cloud SDK using the supplied credentials file and project id. Listing 3 demonstrates the ansible script to implement configure operation.
- **Start** – The start operation fetches the input and output parameters and values from *SourceDF* and *DestinationDF* node types. The io arguments are then combined with the supplied `template_specific_parameters`, if any. The start operation then executes the Dataflow template generated from the Python code using the create operation. The start operation also passes configured io

arguments and supplied arguments while executing the template. Listing 4 demonstrates the ansible script to implement start operation.

- **Stop** – The stop operation is responsible for the cancelation of started Dataflow job. Listing 5 demonstrates the ansible script to implement stop operation.
- **Delete** – The delete operation deletes the temporary files that were created to fetch io arguments from *SourceDF* and *DestinationDF* node types. Listing 6 shows the ansible script to implement delete operation

(B) ***GoogleDataFlowTemplateExecution*** - This node type does not require a Python code for execution. It instead uses Google-provided templates to execute generic data pipeline jobs in Google Dataflow. Each of these templates requires users to provide specific runtime parameters for execution. The list of parameters to be supplied for each template can be found in the template description <sup>6</sup>. Users can provide the input and output parameters using the matching implementation for *SourceDF* and *DestinationDF*. Additional parameters are provided via the **template\_specific\_parameters** property of the *GoogleDataFlowTemplateExecution* node type.

- **Configure** - The configure life-cycle operation for this node type is similar to the configure operation in *GoogleDataFlowPythonExecution* node type and is responsible for configuring the Google Cloud SDK using the supplied credentials file and project id.
- **Start** – The start operation is also similar to the start operation in *GoogleDataFlowPythonExecution* node type. However, the only difference is that it triggers the Dataflow job execution on the supplied template location since this node type does not need to generate template files from Python codes.
- **Stop** – The stop operation is responsible for the cancelation of started Dataflow job. This is also similar to the stop operation in *GoogleDataFlowPythonExecution* node type.
- **Delete** – The delete operation deletes the temporary files that were created to fetch io arguments from *SourceDF* and *DestinationDF* node types. *GoogleDataFlowPythonExecution* node type has a similar ansible script to implement delete operation.

---

```
---
- hosts: localhost
  environment:
```

---

<sup>6</sup><https://cloud.google.com/dataflow/docs/guides/templates/provided-templates>



```

GOOGLE_APPLICATION_CREDENTIALS: "{{ credential_file_path }}"
vars:
  ansible_python_interpreter: /usr/bin/python3
tasks:
  - name: "Install pypi_dependencies : {{ pypi_dependencies }}"
    pip:
      name: "{{ item }}"
      state: present
    loop: "{{ pypi_dependencies }}"

  - name: Generate DataFlow template from python script
    command: python3 {{ dataflow_python_code_path }} --runner
      DataflowRunner --project {{ project_id }} --staging_location
      {{ staging_location }} --temp_location {{ staging_location
      }}/temp --template_location {{ template_location }} --region
      {{ region }}

```

---

Listing 2. Create ansible script for GoogleDataFlowPythonExecution node type

---

```

---
- hosts: localhost
vars:
  ansible_python_interpreter: /usr/bin/python3
tasks:

  - name: Configure authentication for Google Cloud SDK
    shell: gcloud auth activate-service-account --key-file={{
      credential_file_path }}

  - name: Set DataFlow project for Google Cloud SDK
    shell: gcloud config set project {{ project_id }}

```

---

Listing 3. Configure ansible script for GoogleDataFlowPythonExecution node type

---

```

---
- hosts: localhost
vars:
  ansible_python_interpreter: /usr/bin/python3
tasks:

  - name: Verify if IO argument file exists
    stat:
      path: ~/tmp/{{ source_node_id }}/io-args.txt
    register: check_result

  - name: Execute the Google Cloud DataFlow with Input Output
    arguments when additional parameters exists.
    shell: gcloud dataflow jobs run {{ dataflow_job_name }} --gcs-
      location {{ template_location }} --region {{ region }} --

```

```

        staging-location {{staging_location}} --parameters {{
        lookup('file', '~/tmp/{{source_node_id}}/io-args.txt')
        }},{{template_specific_parameters}} --format=json
register: result
when: (check_result.stat.exists) and (
        template_specific_parameters != "None")

- name: save the data to a Variable as a Fact when Input Output
  arguments and additional parameters both exist.
set_fact:
  pipeline_id: "{{ (result.stdout | from_json).id }}"
when: (check_result.stat.exists) and (
        template_specific_parameters != "None")

- name: Execute the Google Cloud DataFlow with Input Output
  arguments when additional parameters does not exist.
shell: gcloud dataflow jobs run {{dataflow_job_name}} --gs-
      location {{template_location}} --region {{region}} --
      staging-location {{staging_location}} --parameters {{
      lookup('file', '~/tmp/{{source_node_id}}/io-args.txt') }}
      --format=json
register: result
when: (check_result.stat.exists) and (
        template_specific_parameters == "None")

- name: save the data to a Variable as a Fact when Input Output
  arguments exist and additional parameters does not exist.
set_fact:
  pipeline_id: "{{ (result.stdout | from_json).id }}"
when: (check_result.stat.exists) and (
        template_specific_parameters == "None")

- name: Execute the Google Cloud DataFlow without any Input or
  Output arguments but with additional parameters.
shell: gcloud dataflow jobs run {{dataflow_job_name}} --gs-
      location {{template_location}} --region {{region}} --
      staging-location {{staging_location}} --parameters {{
      template_specific_parameters}} --format=json
register: result
when: (not check_result.stat.exists) and (
        template_specific_parameters != "None")

- name: save the data to a Variable as a Fact when Input Output
  arguments does not exist and additional parameters exist.
set_fact:
  pipeline_id: "{{ (result.stdout | from_json).id }}"
when: (not check_result.stat.exists) and (
        template_specific_parameters != "None")

```

```

- name: Execute the Google Cloud DataFlow without any Input or
  Output arguments or additional parameters.
  shell: gcloud dataflow jobs run {{dataflow_job_name}} --gcs-
    location {{template_location}} --region {{region}} --
    staging-location {{temp_location}} --format=json
  register: result
  when: (not check_result.stat.exists) and (
    template_specific_parameters == "None")

- name: save the data to a Variable as a Fact when neither
  Input Output arguments exist nor additional parameters.
  set_fact:
    pipeline_id: "{{ (result.stdout | from_json).id }}"
  when: (not check_result.stat.exists) and (
    template_specific_parameters == "None")

# Set attribute "pipelineID"
- name: set attributes
  set_stats:
    data:
      dataflow_job_id: "{{ pipeline_id }}"

```

---

Listing 4. Start ansible script for GoogleDataFlowPythonExecution node type

---

```

---
- hosts: localhost
  vars:
    ansible_python_interpreter: /usr/bin/python3
  tasks:

    - name: Cancel the running Job
      shell: gcloud dataflow jobs cancel {{pipeline_id}}

```

---

Listing 5. Stop ansible script for GoogleDataFlowPythonExecution node type

---

```

---
- hosts: localhost
  vars:
    ansible_python_interpreter: /usr/bin/python3
  tasks:

    - name: Delete temp content & directory for io-args.
      file:
        state: absent
        path: ~/tmp/{{ source_node_id }}

```

---

Listing 6. Delete ansible script for GoogleDataFlowPythonExecution node type

---

### ***DestinationDF***

This node type is responsible for collecting user output parameters, if any, for the Google

Dataflow job. The *DestinationDF* node types connect to *MidwayDF* with a many-to-many relation, meaning multiple *MidwayDF* node types may provide an outgoing connection to a single *DestinationDF*. It contains three types of node type implementations, which are *GCSBucketOutputDF*, *GCPubSubTopicOutputDF*, and *GCBigqueryTableOutputDF* for different types of sinks. These node types collect the output parameter name and the output location and supply it to *MidwayDF* for its Dataflow job execution. Similar to *SourceDF*, The *DestinationDF* node types do not implement life cycle operations. The available *DestinationDF* node types are discussed below.

- (A) ***GCSBucketOutputDF*** - This node type is used to supply output GCS bucket path for the Dataflow. This node also receives the parameter name, which is described as runtime argument in Dataflow Python code or Dataflow template.
- (B) ***GCPubSubTopicOutputDF*** – This node type receives output Topic to direct the output Dataflow stream. It also receives the parameter name, which is described as a runtime output argument in Dataflow Python code or Dataflow template.
- (C) ***GCBigqueryTableOutputDF*** – This node type is used to receive output Bigquery table for the Dataflow.

### ***GoogleDataflowPlatform***

*GoogleDataflowPlatform* node type does not derive from *PipelineBlock*. However, it is equally essential to execute Google Dataflow using RADON. This node type implements life cycle operations that install Apache Beam library and Google Cloud SDK on the host platform, which is a must to communicate with the Google Cloud platform and template conversion. As this node type installs the prerequisite for the Dataflow execution, *MidwayDF* should only start its lifecycle operation after the complete deployment of this node type. In terms of the TOSCA standard, the *MidwayDF* is hosted on *GoogleDataflowPlatform*.

### **Why separate Abstract TOSCA node types for Dataflow**

Figure 4 demonstrates that a similar abstract node types are created named as *SourceDF*, *MidwayDF*, and *DestinationDF*. One may question why not use the same abstract node types created for existing RADON data pipelines, namely *SourcePB*, *MidwayPB*, and *DestinationPB*. This design decision to create different abstract node types for the Google Dataflow can be attributed to the following reasons.

- The *SourcePB*, *MidwayPB*, and *DestinationPB* are special nodes created to be hosted on the Apache Nifi environment. Adding new node types for Google Dataflow under the same abstract node types may cause future conflicts. Future conflicts may arise when a developer intends to implement a common life-cycle

operation for either the source, midway, or destination Apache Nifi nodes. Developers may not be able to add common Nifi life-cycle operations in Source, Midway, or Destination blocks if Dataflow node types are placed under the same abstract types.

- Currently, *SourcePB* can directly connect to *DestinationPB* for the existing RADON data pipeline. However, for the node types in Dataflow, the Midway block is the core of the Dataflow implementation, and hence Midway block can not be skipped. Obviously, exceptions could be added to reiterate the importance of the Midway block only in the case of Google Dataflow. However, that seems not an optimal solution.

#### 4.2.2 Developed TOSCA relationship and capability types for Google Dataflow

The *SourceDF*, *MidwayDF*, and *DestinationDF* may be connected in the same specified sequence to design a Google Dataflow in the RADON ecosystem. These connections arise from the requirements and capabilities defined for each node type. The requirements and corresponding capabilities for each node type are listed below.

- A *SourceDF* node type has the Requirement to connect to one or more *MidwayDF* node types. This Requirement is satisfied by the Capability *MidwayDF* provides to connect zero or more *SourceDF* node types.
- Similarly, a *MidwayDF* node type has the Requirement to connect to zero or more *DestinationDF*, which is satisfied by the Capability *DestinationDF* provides to connect to one or more *MidwayDF*.
- A *MidwayDF* node type also has the Requirement to be hosted on the *GoogleDataflowPlatform* node type. In return, *GoogleDataflowPlatform* provides the Capability to host *MidwayDF*.

The connection between the Requirements and corresponding Capabilities are made using a specific Relationship type and specific Capability type. TOSCA provides a set of Relationship types and Capability types to describe the requirements and capabilities of node types. One can also create custom relationship types and capability types derived from existing ones. The developed relationship types for Google Dataflow are described below.

##### ***ConnectDataflowInput***

This relationship connects the *SourceDF* node type with *MidwayDF*. It is derived from the TOSCA's base relationship *ConnectsTo*. *ConnectDataflowInput* relationship type implements the *pre\_configure\_target* operation using Ansible script in the Configure interface of the *ConnectsTo* relationship type. *SourceDF* is the source in this relationship, and *MidwayDF* is the target. Based on the TOSCA standard, the target is always

deployed first, which means *MidwayDF* gets deployed first. However, *MidwayDF* needs the input parameters from *SourceDF* to start the dataflow job. As the name suggests, the *pre\_configure\_target* operation runs right before the configure method for the target, *MidwayDF*. The *pre\_configure\_target* operation takes the input parameters from the *SourceDF* and saves them into a host system file before the start operation runs for *MidwayDF*. The same file holds the output parameters too. The start operation for the *MidwayDF* then starts the Dataflow job using the input and output parameters from the file. The ansible script that implements the *pre\_configure\_target* operation is shown in Listing 7

---

```

---
- hosts: localhost
  vars:
    ansible_python_interpreter: /usr/bin/python3
  tasks:

    - name: Verify if IO argument file exists
      stat:
        path: ~/tmp/{{ source_node_id }}/io-args.txt
      register: check_result

    - name: Create temp directory to store input args
      file:
        path: ~/tmp/{{ source_node_id }}
        state: directory
      when: not check_result.stat.exists

    - name: Create io-args.txt file to store first input argument
      for the DataFlow
      lineinfile:
        path: ~/tmp/{{ source_node_id }}/io-args.txt
        line: '{{ param_name }}={{ input }}'
        create: yes
      when: not check_result.stat.exists

    - name: Append additional input argument to the io-args.txt
      file
      lineinfile:
        path: ~/tmp/{{ source_node_id }}/io-args.txt
        regexp: '^(.*)$'
        line: '\\1,{{ param_name }}={{ input }}'
        backrefs: yes
      when: check_result.stat.exists

```

---

Listing 7. Ansible script to implement *pre\_configure\_target* operation for *ConnectDataflowInput* relationship type

### ***ConnectDataflowOutput***

This relationship connects *MidwayDF* with *DestinationDF*. Similar to *ConnectDataflowInput*, this relationship also derives from the TOSCA's base relationship *ConnectsTo* and implements the *pre\_configure\_source* operation defined in the Configure interface. The *pre\_configure\_source* operation runs right before the configure operation of the source *MidwayDF*. This operation fetches the output parameters from the *DestinationDF* and saves them into a host system file where input parameters will also be saved. The start operation then starts the Dataflow job along with the output parameters. The ansible script that implements the *pre\_configure\_source* operation is shown in Listing 8

---

```
---
- hosts: localhost
  vars:
    ansible_python_interpreter: /usr/bin/python3
  tasks:

    - name: Verify if IO argument file exists
      stat:
        path: ~/tmp/{{ source_node_id }}/io-args.txt
      register: check_result

    - name: Create temp directory to store IO args
      file:
        path: ~/tmp/{{ source_node_id }}
        state: directory
      when: not check_result.stat.exists

    - name: Create io-args.txt file to store first output argument
      for the DataFlow
      lineinfile:
        path: ~/tmp/{{ source_node_id }}/io-args.txt
        line: '{{ param_name }}={{ output }}'
        create: yes
      when: not check_result.stat.exists

    - name: Append additional output argument to the io-args.txt
      file
      lineinfile:
        path: ~/tmp/{{ source_node_id }}/io-args.txt
        regexp: '^(.*)$'
        line: '\\1,{{ param_name }}={{ output }}'
        backrefs: yes
      when: check_result.stat.exists
```

---

Listing 8. Ansible script to implement *pre\_configure\_source* operation for *ConnectDataflowOutput* relationship type

This thesis has also designed specific Capability types for the Google Dataflow. The developed capability type is discussed below.

#### ***ConnectToDataFlow***

This capability type is derived from the TOSCA base Capability type *Endpoint*. The *MidwayDF* and *DestinationDF* provide this Capability to connect *SourceDF* and *MidwayDF*, respectively. This developed Capability does not hold any specific implementation different from its parent type and is just used to represent the Capability type for Google Dataflow node types.

#### **Designing Service Template using TOSCA components for Dataflow**

Developers can use RADON graphical modeling tool (Winery) to design Google Dataflow as a Service Template using developed components. Users must first select the appropriate *SourceDF* and *DestinationDF* node types based on the required input and output type. Users then need to select either the *GoogleDataFlowTemplateExecution* or *GoogleDataFlowTemplateExecution* node type as their *MidwayDF*. Users may select the node type based on their need for Dataflow creation via Python code or Google-provided templates. After selecting the node types, Users need to connect the *SourceDF*, *MidwayDF*, and *DestinationDF* in a sequence according to the requirements and capabilities for each node type. The next step involves users supplying the required properties and artifacts to the node types involved and finally exporting the Service Template as a CSAR file. RADON orchestrator can then deploy and manage the Google Dataflow as per user need. Section 5 contains detailed examples for designing Google Dataflow using Winery.

### **4.3 Interaction between Dataflow and developed TOSCA components**

Figure 5 provides a high-level architecture of how the TOSCA components for Dataflow communicate with the Google Dataflow platform for automated orchestration. The user may provide the absolute path as the TOSCA property for a *MidwayDF* node type. The user also provides input and output location for the job by setting the property value in the *InputDF* and *OutputDF* node types. The user may also add multiple Input and Output components to provide multiple IO arguments. The supplied IO arguments must be consistent with the runtime arguments expected in the written Python code.

The RADON orchestrator creates the executable template from the Python code; it then sends a request to Google Dataflow to start the job. The request contains the metadata for job execution, such as the location of the executable template, runtime parameters, and region. In response, the Google Dataflow returns the job id, which the RADON orchestrator can use to stop the Dataflow job later.



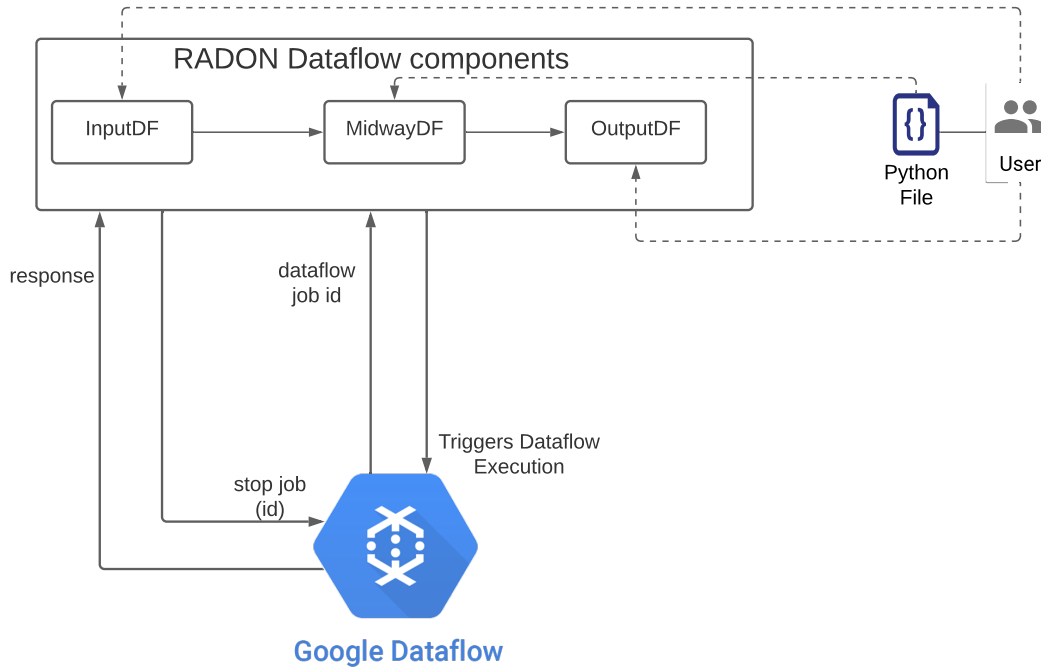


Figure 5. RADON Dataflow implementation

#### 4.4 Integrating Apache Nifi and Google Dataflow

The existing TOSCA components for Apache Nifi data pipeline and newly developed node types for Google Dataflow may be used in conjunction to design data pipelines that stretch over the private cloud, public cloud resources, and Google Dataflow. Figure 6 shows how the existing RADON data pipeline can connect to the newly developed Google Dataflow block. The output of the Apache Nifi data pipeline may be supplied as the input to the Google Dataflow and vice versa. One point to note is that the implemented Google Dataflow only supports Google-based services for input and output. Therefore, only Google-based sources and sinks can be connected from the RADON data pipeline to Google Dataflow node types. Developers can terminate their Apache Nifi data pipeline using *DestinationPB* that supports publishing data to a Google-based service. Thereby, *SourceDF* node type from Google Dataflow can read the input from the exact location and extend the data pipeline further. Similarly, As Google Dataflow may finish the job by writing the data to a Google-based service, the *SourcePB* is configured to read the data from the exact Google-based service.

The connection between the Apache Nifi based data pipeline and Google Dataflow node types are made using newly developed Relationship types and Endpoint Capability.

The developed relationship types to allow integration are described below.

- **ConnectDataflowToNifi** - This relationship connects the *DestinationDF* node type with *SourcePB*. It is derived from the TOSCA's base relationship *ConnectsTo* and does not implement any interface or operation. Users must use this relationship type to connect the appropriate node type in *DestinationDF* to match the node type in *SourcePB*. For, E.g., *GCSBucketOutputDF* only connects to matching *ConsGCSBucket* nodes because both the node types deal with GCS Buckets.
- **ConnectNifiToDataflow** - This relationship connects the *DestinationPB* node type with *SourceDF*. It is derived from the TOSCA's base relationship *ConnectsTo* and does not implement any interface or operation. Users must use this relationship type to connect the appropriate node type in *DestinationPB* to match the node type in *SourceDF*. For, E.g., *PubGCS* only connects to matching *GCSBucketInputDF* nodes because both the node types deal with GCS Buckets.

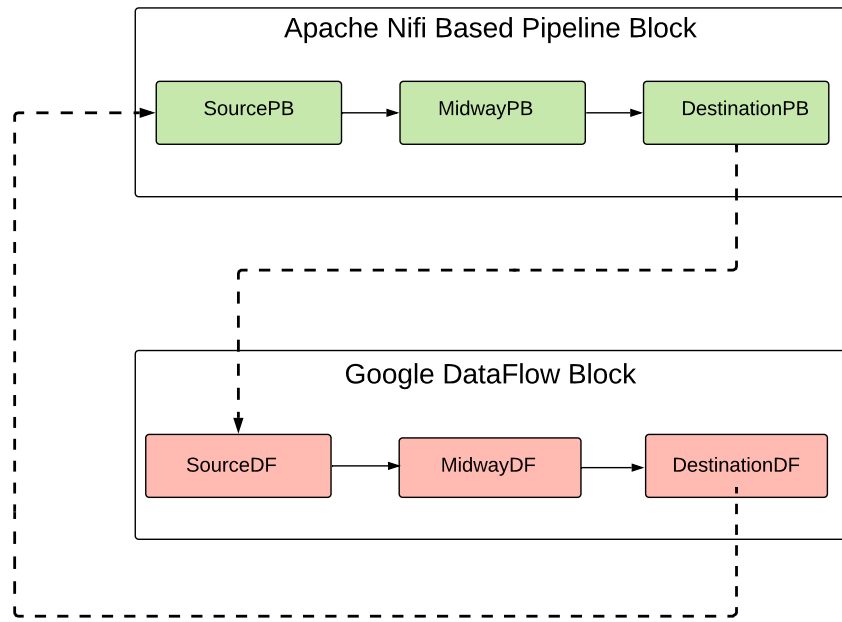


Figure 6. Integration of Google Dataflow with RADON data pipeline

Section 5.4 presents an example of a data pipeline that starts with the Google Dataflow processing the image and publishing the result to a GCS bucket. Apache Nifi based RADON data pipeline extends by reading the input from the same GCS bucket and publishing it to an AWS S3 bucket.

## 5 Evaluation and Results

This section discusses the evaluation of the developed TOSCA components for Google Dataflow by deploying them as a Service Template to orchestrate batch and streaming Dataflow. This section validates the xOpera deployment results for the developed node types by verifying the successful Dataflow job creation in the Google Cloud Console. The successful execution of the Dataflow job is validated by verifying the xOpera deploy and undeploy logs and analyzing the generated Dataflow job DAG and final output resulting from the Dataflow job. Similarly, integration of Dataflow node types with Apache Nifi based existing node types is validated by designing a Service Template for a batch processing data pipeline that starts with Dataflow and ends with Apache Nifi publishing the final result. The xOpera deployment results for the integrated Service Template are validated by confirming the successful Dataflow job creation in Cloud Console and Apache Nifi and the Processor groups creation. The Cloud buckets are checked for the intermediate and final output files expected as result of data pipeline execution.

All the input and output files and figures used in this section can be found in the GitHub Repository. Please refer to the GitHub Repository to access complete figures and files[19].

### Example Descriptions

- (A) **Example 1:** This example was chosen to demonstrate a template-based Dataflow deployment for Streaming data using developed TOSCA components. The used template is Google-provided public templates for performing general tasks. The designed Dataflow receives input from a Google Pub/Sub subscription and copies the same data to a Google BigQuery table. Hence, this example uses one InputDF and one OutputDF node type. Also, it uses Google provided template to design the Dataflow. The example is illustrated in detail in section 5.2.
- (B) **Example 2:** This goal of this example is to demonstrate a python-based Dataflow deployment for batch data processing using developed TOSCA components. The designed Dataflow receives input from three locations in Google Cloud Storage and writes the output to two locations in Google Cloud Storage. Hence, three InputDF node types and two OutputDF node types are used. This Google Dataflow counts the frequency of all the words in the input files. This example is discussed in detail in section 5.3.
- (C) **Example 3:** This example was chosen to illustrate the integration between Google Dataflow node types and Apache Nifi node types. The designed data pipeline first uses the Google Dataflow to process the image in this example. It later uses Apache Nifi based data pipeline to copy the data from the GCS bucket to the AWS

S3 bucket. The Google Dataflow takes one *InputDF* node type to receive GCS bucket input and then writes the processed image to another GCS bucket using one *OutputDF* node type. Python code is used to describe the workflow for this Google Dataflow. Similarly, for Apache Nifi based data pipeline, a *ConsGCSBucket* is used to read the image files from the GCS bucket, and *PubsS3Bucket* is used to copy the image files to the S3 bucket. This example is discussed in detail in section 5.4.

## 5.1 Evaluation Setup

In order to validate the developed RADON components, this section sets up the environment to orchestrate example data pipelines. A required set of steps for performing this evaluation are listed below.

- An OpenStack host with a Ubuntu 18.04 image and m3.tiny flavor with 2 vCPUs, 2 GB of RAM, and 10 GB of total disk was configured.
- RADON graphical modeling tool Winery with version 'Winery 1.0.0-SNAPSHOT' was installed on the OpenStack host.
- RADON orchestrator xOpera was installed on the OpenStack host.
- A Google cloud project was created, and free tier billing was enabled for the same.
- Google Cloud APIs for Dataflow, Google Cloud Storage, BigQuery, Cloud Pub/Sub, and all its dependencies were enabled as instructed in Google documentation[15].
- An AWS account with free tier was created.

The above-mentioned resources and platform setup are common for all three example data pipelines. Appendix III provides a Technical Manual that describes the above steps in detail. It explains all the installation steps, including the platform setup needed to deploy the RADON data pipeline explained in section 5.3.

## 5.2 Pub/Sub Subscription to BigQuery using Dataflow

This section orchestrates a Google Dataflow job for streaming data using a Google-provided template. This example intends to demonstrate deploying and managing Google Dataflow jobs for streaming data in the RADON ecosystem. In this example, Dataflow reads the JSON-formatted streaming data from a Google Pub/Sub Subscription and publishes them to a Google Bigquery table. Google Pub/Sub <sup>7</sup> is a publish-subscribe

---

<sup>7</sup><https://cloud.google.com/pubsub>

message pattern in which publishers publish asynchronous messages to a topic. Systems may create one or more subscriptions for a particular topic to consume those published messages. Whereas Google BigQuery <sup>8</sup> is a serverless data warehouse solution for enterprises that supports SQL queries. Figure 7 illustrates the high-level architecture of the designed Google Dataflow.

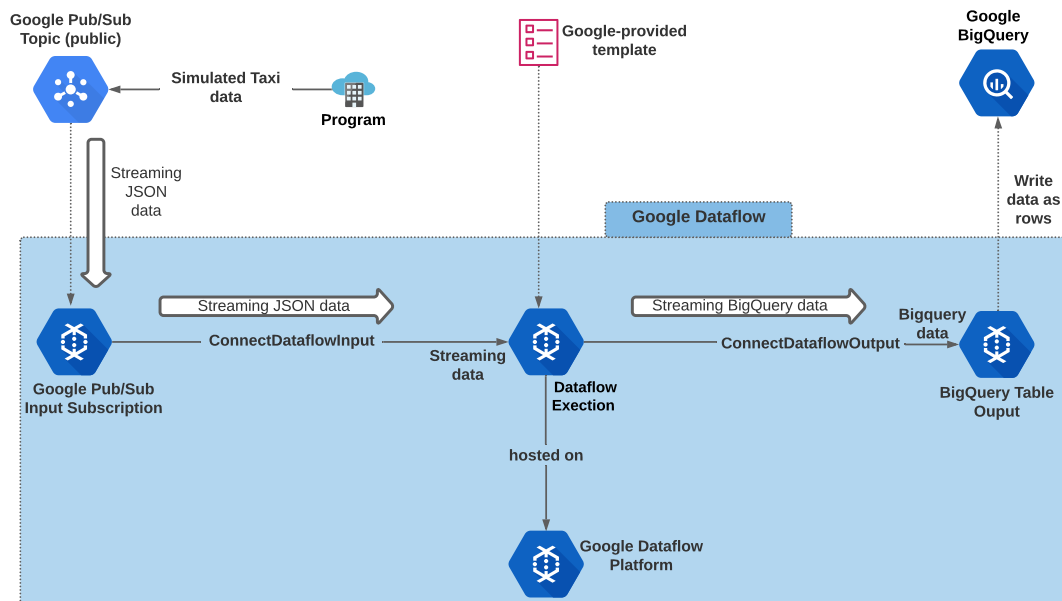


Figure 7. High-level architecture of Pub/Sub Subscription to BigQuery Dataflow

### Input Streaming Data

The Google-provided public Pub/Sub topic receives simulated streaming data for New York taxi rides, which then forwards the data to the created Pub/Sub Subscription. Listing 9 shows the sample JSON data received by the created Pub/Sub subscription. The velocity of the data received by the Subscription is extremely high, meaning thousands of JSON records are sent to the Pub/Sub subscription every few seconds. This high velocity of incoming data means thousands of records are written to the Bigquery table every few seconds. While deploying this example Dataflow, one must exercise caution to avoid exceeding free tier limits.

<sup>8</sup><https://cloud.google.com/bigquery>

### 5.2.1 Prerequisites for this Dataflow

Because this Dataflow job involves reading data from a Pub/Sub subscription and writing it to a Bigquery table, Pub/Sub Subscription and a Bigquery table are created. Details are described below:

- **Pub/Sub subscription** - This example creates a Pub/Sub subscription named **taxiRidePublicDataSubscription** to the topic 'projects/pubsub-public-data/topics/taxirides-realtime'<sup>9</sup>. The configured topic is a publically available topic receiving emulated taxi data stream with a fixed JSON schema. Listing 9 shows a sample JSON message that the Subscription receives from the topic.

---

```
{
  "ride_id": "cd13c679-a642-43a5-9d94-35e8f81993b0",
  "point_idx": 17,
  "latitude": 40.77552,
  "longitude": -73.98227,
  "timestamp": "2021-12-24T00:15:23.09698-05:00",
  "meter_reading": 0.66539377,
  "meter_increment": 0.03914081,
  "ride_status": "enroute",
  "passenger_count": 5
}
```

---

Listing 9. Sample JSON message

- **BigQuery table** - This example creates a Bigquery table named **taxiRidesSampleData** inside the BigQuery dataset **myRadonExampleDataSet**, so that the Dataflow can write the output messages to it. The Bigquery table schema must match the JSON schema. Figure 8 shows the schema for the Bigquery table, which must match the incoming JSON schema from Pub/Sub subscription.

### 5.2.2 Choosing the appropriate node types and supplying properties

Since this example uses a Google-provided template, developers must refer to the Google Dataflow documentation<sup>10</sup> to get the GCS location for the template and its parameters. In this case, the template has three mandatory and two optional parameters:

- **inputSubscription** - The Pub/Sub subscription to read input. This parameter will be provided via the *GCPubSubSubscriptionInputDF* node type.

---

<sup>9</sup><https://github.com/googlecode/clouddataflow-nyc-taxi-tycoon>

<sup>10</sup><https://cloud.google.com/dataflow/docs/guides/templates/provided-streaming#pubsub-subscription-to-bigquery>

Field name	Type	Mode
ride_id	STRING	NULLABLE
point_idx	INTEGER	NULLABLE
latitude	FLOAT	NULLABLE
longitude	FLOAT	NULLABLE
timestamp	TIMESTAMP	NULLABLE
meter_reading	FLOAT	NULLABLE
meter_increment	FLOAT	NULLABLE
ride_status	STRING	NULLABLE
passenger_count	INTEGER	NULLABLE

Figure 8. Table schema for BigQuery table taxiRidesSampleData

- **outputTableSpec** - The BigQuery output table location. This parameter will be provided via *GCBigqueryTableOutputDF* node type.
- **outputDeadletterTable** - Another Bigquery table for the messages that failed to reach the output table. If this table does not exist, it is created during Dataflow execution. The developer must provide this parameter using the `template_specific_paramter` property inside the *GoogleDataFlowTemplateExecution* node type.
- **javascriptTextTransformGcsPath (optional)** - If needed, developers may provide these non-IO parameters as `template_specific_paramter` property inside *GoogleDataFlowTemplateExecution* node type.
- **javascriptTextTransformFunctionName (optional)** - If needed, developers may provide these non-IO parameters as `template_specific_paramter` property inside *GoogleDataFlowTemplateExecution* node type.

Also, due to the use of a Google-provided template for job execution, developers must choose *GoogleDataFlowTemplateExecution* as their *MidwayDF*. Chosen node types and

their supplied properties are discussed below.

### ***GCPubSubSubscriptionInputDF***

*GCPubSubSubscriptionInputDF* accepts two required properties. It also accepts optional properties related to scheduling. However, the node types do not support scheduling yet. Still, the scheduling-related properties may come in handy after the scheduling is implemented for Google Dataflow. Table 1 describes the configured properties for this node type.

Property	Value	Description
input	projects/thesis-project-329917/subscriptions/taxiRidePublicDataSubscription	This is the Pub/Sub subscription for reading the stream, and has the format of projects/<project_id>/subscriptions/<subscription <sub>i</sub> d >
param_name	inputSubscription	This is the parameter name expected by the template for the 'input' property provided above. The parameter name can be found in the template specification.
schedulingStrategy (optional)	EVENT_DRIVEN	Leave the value to default.
name (optional)	inputSubscriptionNode	Name of the pipeline node
schedulingPeriod-CRON (optional)	* * * * *	Leave the value to default.

Table 1. Properties for *GCPubSubSubscriptionInputDF* node type

### ***GoogleDataFlowTemplateExecution***

*GoogleDataFlowTemplateExecution* node type is the central processing hub for the RADON Dataflow. This node type triggers the actual job execution after receiving the input and the output from other node types. Below, Table 2 describes the essential configured properties for this node type.



Property	Value	Description
template_location	gs://dataflow-templates/latest/PubSub_Subscription_to_BigQuery	This is the GCS location where the Dataflow template is stored. Since a Google-provided template is used, developers can get the location of this template from Dataflow documentation.
template_specific_parameters	outputDeadletterTable=thesis-project-329917:myRadonExampleDataSet.taxiRidesSampleDataDL	This is the template-specific parameter to supply additional parameters needed for a template. Here, BigQuery table taxiRidesSampleDataDL is supplied in the format parameter_name=value. Multiple parameters can be separated with a comma.
dataflow_job_name	PubSubToBigQueryJob	The name of the Google Dataflow job.
project_id	thesis-project-329917	The Google project id where the Dataflow job will run
credential_file_path	/home/ubuntu/dp-project/cred/thesis-project-329917-5c40808bbb3e.json	The system path to the Service Account JSON key. It provides authorization for the Dataflow and the configured project. .
staging_location	gs://dp-bucket-1/staging	Google Cloud location where developer want to store the temporary and intermediate files. .
region	europe-west1	This is the cloud region where developer want to execute the Dataflow job. .

Table 2. Properties for GoogleDataFlowTemplateExecutionTable node type

### ***GCBigqueryTableOutputDF***

*GCBigqueryTableOutputDF* accepts two mandatory properties. It also accepts optional properties related to scheduling. However, the Google Dataflow node types do not support scheduling yet. Therefore, these scheduling-related properties for the future when scheduling has been implemented. Table 3 describes the mandatory properties for this node type.

Property	Value	Description
output	thesis-project-329917:myRadonExampleDataSet.taxiRidesSampleData	This is the BigQuery table name where the Dataflow will publish the data, and has the format of <project>:<dataset>.<my-table>
param_name	outputTableSpec	This is the parameter name expected by the template for the 'output' property provided above. The parameter name can be found in the template specification.

Table 3. Properties for GCBigqueryTableOutputDF node type

### ***GoogleDataflowPlatform***

This node type hosts the MidwayDF node type. In this particular example, it hosts the GoogleDataFlowTemplateExecution node type. It does not require any mandatory property to be configured. This node type is responsible for installing the Google Cloud SDK and Apache Beam SDK that enables the host to communicate with Google Dataflow APIs.

### **5.2.3 Designing the service template in the Winery**

This example creates a new Service template named *GooglePubSubSubscriptionToBigqueryDataFlow* to model the Google Dataflow in Winery. Figure 9 shows the Google Dataflow modeling using Winery. Node types described in the previous section are connected using their defined Requirements and Capabilities. Here, this example sets up relationships by performing the below steps:

- Dragging *ConnectDataflowInput* Requirement from *GCPubSubSubscriptionInputDF* into *GoogleDataFlowTemplateExecution*'s *connectToDataflow* Capability.
- Dragging *ConnectToDataflowOutput* Requirement from *GoogleDataFlowTemplateExecution* into *GCBigqueryTableOutputDF*'s *connectToDataflow* Capability.
- Dragging *HostedOn* Requirement from *GoogleDataFlowTemplateExecution* into *GoogleDataflowPlatform*'s *host* Capability.

Next, all the properties discussed in the previous section are configured for the node types using the Winery. Finally, the Service Template design is saved the CSAR file is exported using the Export button in the Winery.

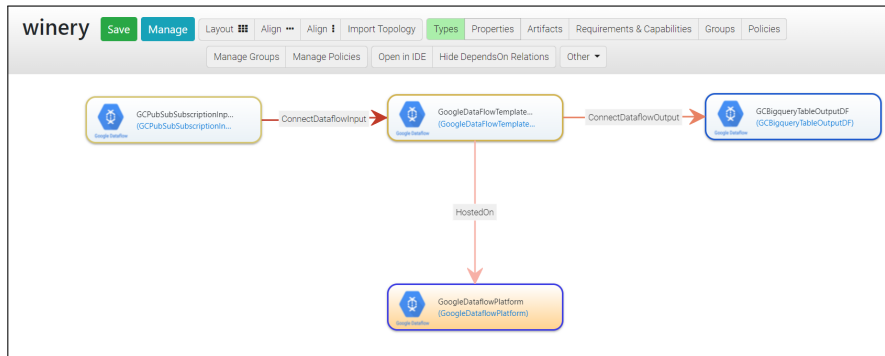


Figure 9. Designing the Service Template *GooglePubSubSubscriptionToBigquery-DataFlow* Winery

#### 5.2.4 Deploying the exported CSAR file using xOpera

RADON orchestrator xOpera requires Cloud Service Archive (CSAR) file to orchestrate the Google Dataflow. this example deploys the exported CSAR file from the previous section using the below command.

```
opera deploy -r GooglePubSubSubscriptionToBigqueryDataFlow.csar
```

Figure 10 shows the Dataflow xOpera deployment logs after deploying the Google Dataflow. After the CSAR file deployment is complete, the Google Dataflow job execution should start.

```
(.venv) ubuntu@my-xopera:~/dp-project$ opera deploy -r GooglePubSubSubscriptionToBigqueryDataFlow.csar
[Worker_0] Deploying GoogleDataFlowPlatform_0_0
[Worker_0]   Executing create on GoogleDataFlowPlatform_0_0
[Worker_0] Deployment of GoogleDataFlowPlatform_0_0 complete
[Worker_0] Deploying GCBigqueryTableOutputDF_0_0
[Worker_0]   Executing create on GCBigqueryTableOutputDF_0_0
[Worker_0]   Executing configure on GCBigqueryTableOutputDF_0_0
[Worker_0]   Executing start on GCBigqueryTableOutputDF_0_0
[Worker_0] Deployment of GCBigqueryTableOutputDF_0_0 complete
[Worker_0] Deploying GoogleDataFlowTemplateExecution_0_0
[Worker_0]   Executing create on GoogleDataFlowTemplateExecution_0_0
[Worker_0]   Executing pre_configure_source on GoogleDataFlowTemplateExecution_0_0--GCBigqueryTableOutputDF_0_0
[Worker_0]   Executing pre_configure_target on GCPubSubSubscriptionInputDF_0_0--GoogleDataFlowTemplateExecution_0_0
[Worker_0]   Executing configure on GoogleDataFlowTemplateExecution_0_0
[Worker_0]   Executing start on GoogleDataFlowTemplateExecution_0_0
[Worker_0] Deployment of GoogleDataFlowTemplateExecution_0_0 complete
[Worker_0] Deploying GCPubSubSubscriptionInputDF_0_0
[Worker_0]   Executing create on GCPubSubSubscriptionInputDF_0_0
[Worker_0]   Executing configure on GCPubSubSubscriptionInputDF_0_0
[Worker_0]   Executing start on GCPubSubSubscriptionInputDF_0_0
[Worker_0] Deployment of GCPubSubSubscriptionInputDF_0_0 complete
```

Figure 10. xOpera deployment logs for Streaming Dataflow

After the Dataflow deployment using xOpera, developer must wait around 8-10 minutes to give Google Dataflow enough time to provision the cloud resources and start the job execution. Then, developer must un-deploy the Dataflow using the below xOpera command.

```
opera undeploy
```

Figure 11 shows the xOpera logs for the undeploy method, which cancels the Dataflow job execution.

```
(.venv) ubuntu@my-xopera:~/dp-project$ opera undeploy
[Worker_0] Undeploying GCPubSubSubscriptionInputDF_0_0
[Worker_0] Executing stop on GCPubSubSubscriptionInputDF_0_0
[Worker_0] Executing delete on GCPubSubSubscriptionInputDF_0_0
[Worker_0] Undeployment of GCPubSubSubscriptionInputDF_0_0 complete
[Worker_0] Undeploying GoogleDataFlowTemplateExecution_0_0
[Worker_0] Executing stop on GoogleDataFlowTemplateExecution_0_0
[Worker_0] Executing delete on GoogleDataFlowTemplateExecution_0_0
[Worker_0] Undeployment of GoogleDataFlowTemplateExecution_0_0 complete
[Worker_0] Undeploying GoogleDataflowPlatform_0_0
[Worker_0] Executing delete on GoogleDataflowPlatform_0_0
[Worker_0] Undeployment of GoogleDataflowPlatform_0_0 complete
[Worker_0] Undeploying GCBigqueryTableOutputDF_0_0
[Worker_0] Executing stop on GCBigqueryTableOutputDF_0_0
[Worker_0] Executing delete on GCBigqueryTableOutputDF_0_0
[Worker_0] Undeployment of GCBigqueryTableOutputDF_0_0 complete
(.venv) ubuntu@my-xopera:~/dp-project$
```

Figure 11. xOpera undeploy logs for Streaming Dataflow

### 5.2.5 Evaluating final results

This example verifies the Dataflow job execution using the Google cloud console after the xOpera deployment in Section 5.1.4. Figure 12 is a screenshot of the Google cloud console that demonstrates the Directed Acyclic Graph (DAG) for the running Dataflow job. The Dataflow DAG in the figure demonstrates the sequence of steps performed for this Dataflow. The DAG starts with *ReadPubSubSubscription* step that reads input from Pub/Sub subscription. Subsequently, *ConvertMessagesToTableRow* step, similar to its name, receives the data from the previous step *ReadPubSubSubscription* and converts the Pub/Sub messages to table rows. The successfully converted messages from the previous step are written to the BigQuery table using the *WriteSuccessfulRecords* step. In contrast, the messages that failed to convert are flattened using *Flatten step* and written to a dead letter table using *WriteFailedRecords*. Similarly, the messages that failed to be written using *WriteSuccessfulRecords* step are wrapped with insertion errors using *WrapInsertionErrors* step and then written to dead letter table using *WriteFailedRecords2*.

This example verifies the BigQuery table in the Google cloud console to ensure that the Dataflow job writes the JSON-formatted Subscription data into the empty BigQuery table. This example uses an SQL query to query the number of rows that the BigQuery table contains after running the job. Figure 13 shows the result of running the count query using the Google cloud console. This example validates the 3229838 rows written where each row corresponds to a JSON object received via the created Subscription. Figure 14 shows the result of an SQL query that queries the rows of stored data. The BigQuery

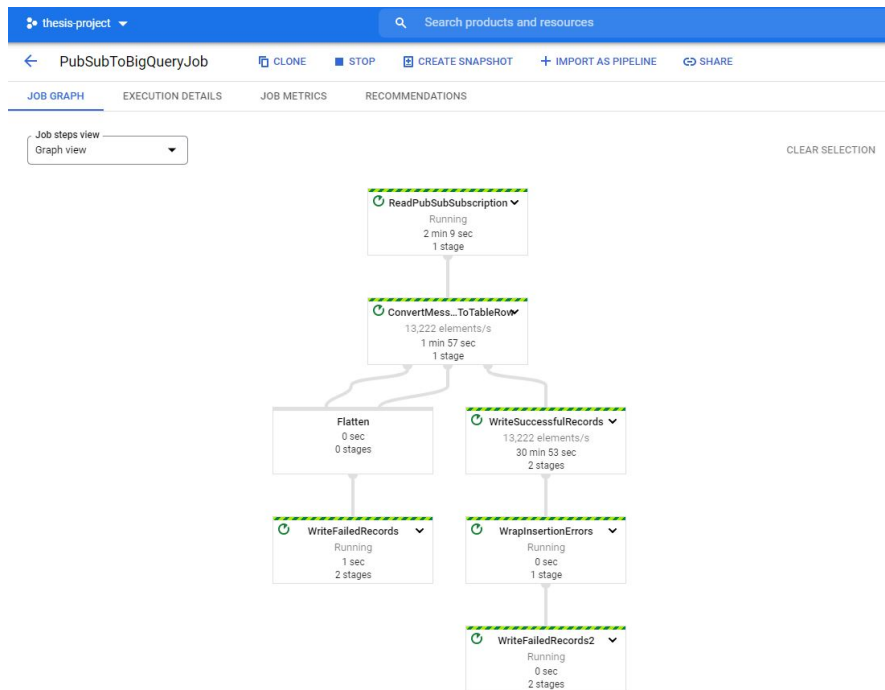


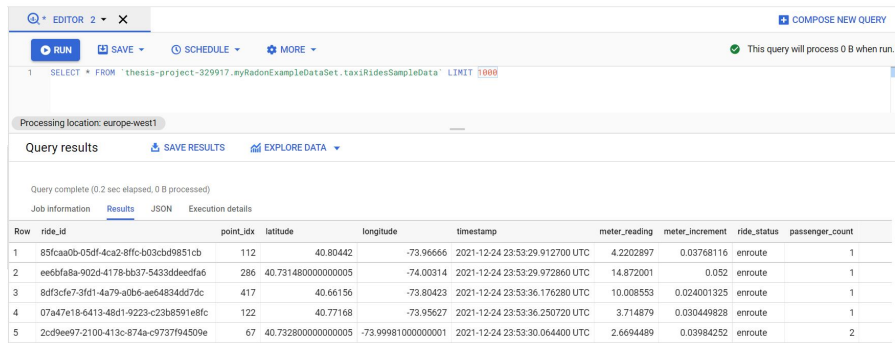
Figure 12. Dataflow Streaming Job workflow in Google cloud console

table records with a limit of 1000 records are exported as a CSV file and placed in the GitHub repository[19] for this thesis.

* EDITOR 2 ✕	
<b>RUN</b>	SAVE SCHEDULE MORE
1 SELECT count(*) FROM `thesis-project-329917.myRadonExampleDataSet.taxiRidesSampleData` LIMIT 1000	
Processing location: europe-west1	
Query results	SAVE RESULTS EXPLORE DATA
Query complete (0.5 sec elapsed, 0 B processed)	
Job information	Results JSON Execution details
Row	f0_
1	3229838

Figure 13. Record counts in the BigQuery table taxiRidesSampleData

Next, this example verifies the Dataflow job execution state in the Google cloud console after running the xOpera un-deployment in previous section. Figure 15 shows



Row	ride_id	point_idx	latitude	longitude	timestamp	meter_reading	meter_increment	ride_status	passenger_count
1	85fcaa0b-05df-4ca2-8ffc-b03cbd9851cb	112	40.80442	-73.96666	2021-12-24 23:53:29.912700 UTC	4.2202897	0.03768116	enroute	1
2	ee6bfaba-902d-4178-bb37-5433ddeedfa6	286	40.7314800000000005	-74.00314	2021-12-24 23:53:29.972860 UTC	14.872001	0.052	enroute	1
3	8df3cf67-3fd1-4a79-a0b6-ae4834d67dc	417	40.66156	-73.80423	2021-12-24 23:53:36.176280 UTC	10.008553	0.024001325	enroute	1
4	07a47e18-6413-48d1-9223-c23b6591e6fc	122	40.77168	-73.95627	2021-12-24 23:53:36.250720 UTC	3.714879	0.030449828	enroute	1
5	2cd9ee97-2100-413c-874a-c9737f94509e	67	40.7328000000000005	-73.9998100000000001	2021-12-24 23:53:30.064400 UTC	2.6694489	0.03984252	enroute	2

Figure 14. Sample records in the BigQuery table taxiRidesSampleData

the state of the Dataflow job after the same. As seen in the figure, each DAG step is shown in the Failed state, meaning the Dataflow job has failed and stopped executing. As expected after the opera un-deploy command, the streaming job is canceled, and no more data is copied from Subscription to the BigQuery table.

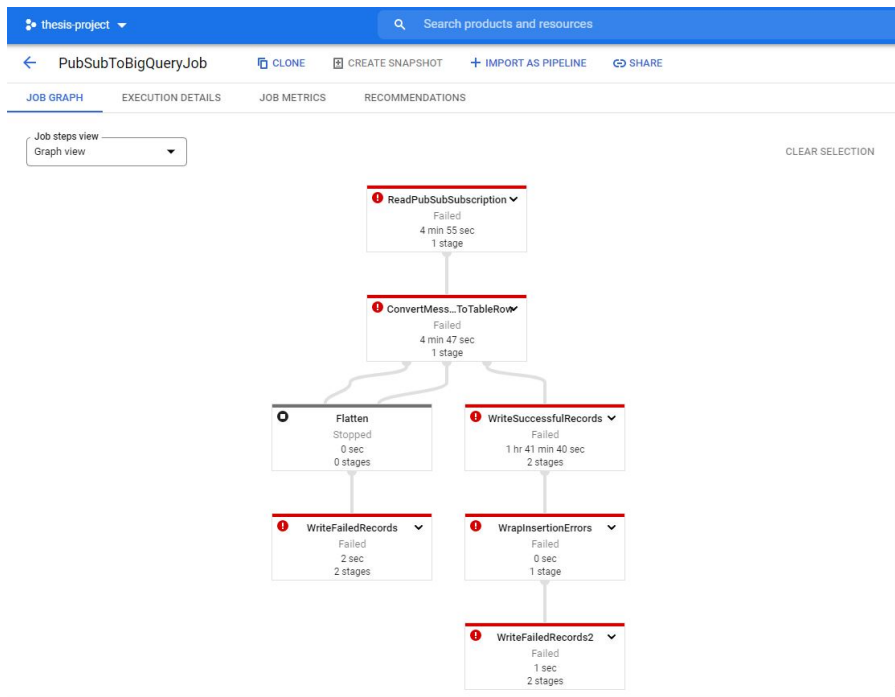


Figure 15. Dataflow Job in Google Cloud Console after opera undeploy

### 5.3 Wordcount example using Dataflow (Batch)

This example orchestrates a Google Dataflow job that reads three different files from three different locations in the GCS bucket, performs the frequency count for the words in those files, and publishes the output result to two different GCS buckets. Python code is written to design this Dataflow, and the designed code uses the Apache Beam library to define the data pipeline. The input and output locations are specified as runtime arguments to the Python code. Therefore, when the MidwayDF create operation generates the corresponding template from the Python code, then the generated template will also require the input and output arguments as a parameter. The previous example supplies input and output parameters using the InputDF and OutputDF node types. Any additional parameters may be supplied via `template_specific_parameters` property in MidwayDF node type. Since Python code is used to define the Dataflow DAG, developers must select *GoogleDataFlowPythonExecution* as the *MidwayDF* node type. Figure 16 shows the high-level architecture for the WordCount example.

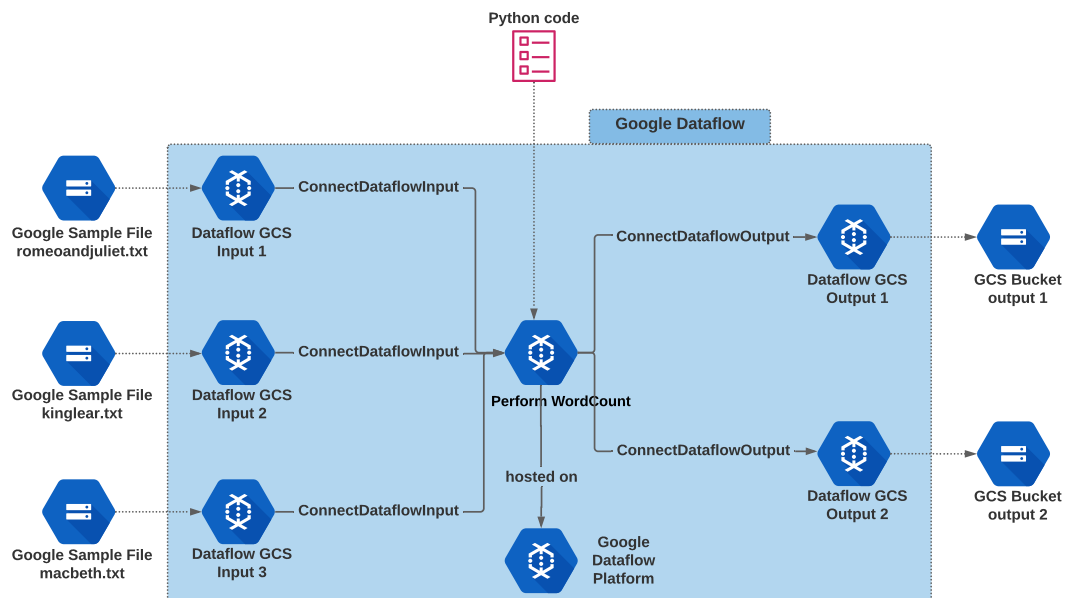


Figure 16. High-level architecture for WordCount Dataflow

Google-provided sample text files are used as an input to this Wordcount example. The workflow for this example Dataflow is described below:

- The Dataflow job reads the first file from the location `gs://dataflow-samples/shakespeare/kinglear.txt`

- The Dataflow reads the second file from the location `gs://dataflow-samples/shakespeare/romeoandjuliet.txt`
- The Dataflow reads the third file from the location `gs://dataflow-samples/shakespeare/macbeth.txt`
- The Dataflow combines the text in all three files and groups them together.
- The Dataflow calculates the frequency for each word in the text and stores it in (key, value) pair format. Where key is a specific word, and value is the total number of occurrences for that word across the combined text.
- Dataflow writes the final result into the GCS Bucket `gs://dp-output1/my_output/`
- Dataflow writes the same result into another GCS Bucket `gs://dp-output2/my_output/`

### **Input Data Files**

The input data files used in this example are `romeoandjuliet.txt`, `kinglear.txt`, and `macbeth.txt`, with each of them, placed in the public GCS location `gs://dataflow-samples/shakespeare`. Each of the input file is around 150 KB and contains plain text inside. The full text files are placed in the GitHub repository and one can access the repository to gain access to full file contents[19].

#### **5.3.1 Prerequisites for this Dataflow**

Since this example reads the text files from the Google-provided sample texts, developers do not need to create input GCS buckets or upload the text files. However, developers must provide GCS location for those sample text files. Google dataflow will then read the text files from the specified GCS buckets. Also, developers must make sure that the specified output buckets are created before running the Dataflow job. developers must also code the Python file for describing the workflow for the Dataflow job.

#### **GCS bucket for outputs**

This example creates two GCS buckets named `dp-output1` and `dp-output2` to store the output file. developers must ensure that the Google Dataflow is authorized to write to the GCS buckets.

#### **Python code using Apache Beam Library**

Listing 10 shows the written Python code which is an adaptation of WordCount example provided by Apache Beam[20]. The class `WordcountOptions` uses the `add_value_provider_argument` to declare the runtime parameters. Therefore, to execute the Dataflow job using the



defined Python code, one must provide all the runtime parameters. The listed Python code illustrates that the runtime parameters are named input1, input2, and input3 for inputs and output1 and output2 for outputs. These parameter names will be supplied as the property param\_names using the appropriate InputDF and OutputDF node types.

This example uses the Python code that follows Apache Beam Programming guide [21]. The guide assists users in understanding the Beam programming model and Beam SDKs. Users can read and understand essential concepts to write effective data pipeline code for Google Dataflows.

---

```
class WordcountOptions(PipelineOptions):
    @classmethod
    def _add_argparse_args(cls, parser):
        parser.add_value_provider_argument('--input1', help='Path of
            the file to read from')
        parser.add_value_provider_argument('--input2', help='Path of
            the file to read from')
        parser.add_value_provider_argument('--input3', help='Path of
            the file to read from')
        parser.add_value_provider_argument('--output1', help='Output
            file to write results to.')
        parser.add_value_provider_argument('--output2', help='Output
            file to write results to.')

class WordExtractingDoFn(beam.DoFn):

    def process(self, element):
        return re.findall(r'[\w\']+', element, re.UNICODE)

logging.getLogger().setLevel(logging.INFO)
pipeline_options = PipelineOptions()

with beam.Pipeline(options=pipeline_options) as p:
    def print_row(element):
        logging.info("Running pipeline:")

    my_options = pipeline_options.view_as(WordcountOptions)

    lines1 = p | 'Read1' >> ReadFromText(my_options.input1)
    lines2 = p | 'Read2' >> ReadFromText(my_options.input2)
    lines3 = p | 'Read3' >> ReadFromText(my_options.input3)

    counts = (
        (lines1, lines2, lines3)
        | 'Flatten' >> beam.Flatten()
        | 'Split' >> (beam.ParDo(WordExtractingDoFn()).
            with_output_types(str))
```

```

| 'PairWithOne' >> beam.Map(lambda x: (x, 1))
| 'GroupAndSum' >> beam.CombinePerKey(sum))

def format_result(word, count):
    return '%s: %d' % (word, count)

output = counts | 'Format' >> beam.MapTuple(format_result)

output | 'Write1' >> WriteToText(my_options.output1)
output | 'Write2' >> WriteToText(my_options.output2)

p.run().wait_until_finish()

```

---

Listing 10. Python Code extracts for WordCount example[20]

### 5.3.2 Choosing the appropriate node types and supplying properties

This example reads the data from three different locations in the GCS bucket; thereby, it use three *GCSBucketInputDF* node types for the input. Also, because Dataflow job is created using the Python code, developers must choose *GoogleDataFlowPythonExecution* node type as the *MidwayDF*. Finally, this example chooses two *GCSBucketOutputDF* node types to represent the output GCS locations. Supplied properties for the chosen node types are described below:

#### ***GCSBucketInputDF***

This example uses three different *GCSBucketInputDF* node types to represent the three inputs, where developers must provide the input file locations and the parameter names via their properties. Since this example Dataflow reads the file from the public GCS bucket, developers must set the input property to the GCS locations to be read. And developers must set the param\_name to input1, input2, and input3 as expected by the Python program.

The mandatory properties and supplied value for the same are listed below for the first *GCSBucketInputDF* node type in Table 4.

Similarly, for the second *GCSBucketInputDF* node type, the mandatory properties and supplied value are described in Table 5. Table 6 describes the properties for the third *GCSBucketInputDF* node type.

Property	Value	Description
input	gs://dataflow-samples/shakespeare/romeoandjuliet.txt	This is the GCS Bucket location for reading the input.
param_name	input1	This is the parameter name expected by the template for the 'input' property provided above. This is defined as IO argument in the Python code.

Table 4. Properties for first GCSBucketInputDF node type

Property	Value	Description
input	gs://dataflow-samples/shakespeare/kinglear.txt	This is the GCS Bucket location for reading the input.
param_name	input2	This is the parameter name expected by the template for the 'input' property provided above. This is defined as IO argument in the Python code.

Table 5. Properties for second GCSBucketInputDF node type

Property	Value	Description
input	gs://dataflow-samples/shakespeare/macbeth.txt	This is the GCS Bucket location for reading the input.
param_name	input3	This is the parameter name expected by the template for the 'input' property provided above. This is defined as IO argument in the Python code.

Table 6. Properties for third GCSBucketInputDF node type

### ***GoogleDataFlowPythonExecution***

Table 7 showcases the properties and the corresponding supplied values for the *GoogleDataFlowPythonExecution* node type.

Property	Value	Description
dataflow_python_code_path	/home/ubuntu/dp-project/myWordCount.py	This is the absolute path to the dataflow python file.
template_location	gs://dp-bucket-1/templates/my-template.json	The Dataflow template generated from the provided Python code is stored at this GCS location.
pypi_dependencies	[]	Provide an empty list as no additional dependencies are needed to execute the supplied Python code.
template_specific_parameters	None	No additional parameters need to be provided for this Dataflow job.
dataflow_job_name	WorCountExampleJob	The name of the Google Dataflow job.
project_id	thesis-project-329917	The Google project id where the Dataflow job will run
credential_file_path	/home/ubuntu/dp-project/cred/thesis-project-329917-5c40808bbb3e.json	The system path to the Service Account JSON key. It provides authorization for the Dataflow and the configured project.
staging_location	gs://dp-bucket-1/staging	Google Cloud location where developers want to store the temporary and intermediate files.
region	europe-west1	This is the cloud region where developers want to execute the Dataflow job.

Table 7. Properties for GoogleDataFlowPythonExecution node type

### ***GCSBucketOutputDF***

This example uses two different *GCSBucketOutputDF* node types to represent the two GCS bucket locations to publish the final resulting file.

developers must set the output property for this node type to the GCS Bucket locations where they want to publish the output data. Moreover, developers must set the param\_name to output1 and output2 as specified in the Python program.

The mandatory properties and supplied value are listed below for the first *GCSBucketOutputDF* node type in Table 8. Similarly, for the second *GCSBucketInputDF* node type, the mandatory properties and supplied value are shown in Table 9.

Property	Value	Description
output	gs://dp-output1/my_output/	This is the GCS Bucket location for writing the output.
param_name	output1	This is the parameter name expected by the template for the 'output' property provided above. This is defined as IO argument in the Python code.

Table 8. Properties for first GCSBucketOutputDF node type

Property	Value	Description
output	gs://dp-output2/my_output	This is the GCS Bucket location for writing the output.
param_name	output2	This is the parameter name expected by the template for the 'output' property provided above. This is defined as IO argument in the Python code.

Table 9. Properties for second GCSBucketOutputDF node type

### ***GoogleDataflowPlatform***

This node type hosts the *GoogleDataFlowTemplateExecution* node type. It does not require any mandatory property to be configured. This node type is responsible for installing the Google Cloud SDK and Apache Beam SDK that enables the host to communicate with Google Dataflow APIs.

### **5.3.3 Designing the service template in the Winery**

A service template named *GoogleDataFlowPythonWordCount* is created using Winery to model the WordCount Google Dataflow job. Figure 17 shows the Google Dataflow modeling using Winery. This example connects the node types involved using their Requirements and Capabilities. The connection steps are described below:

- Dragging *ConnectDataflowInput* requirement from all three *GCSBucketInputDF* node types into the *connectToDataflow* capability of *GoogleDataFlowPythonExecution* node.
- Dragging *ConnectToDataflowOutput* Requirement from *GoogleDataFlowPythonExecution* into *connectToDataflow* Capability for both of the *GCSBucketOutputDF* node types.

- Dragging *HostedOn* Requirement from *GoogleDataFlowPythonExecution* into the *host* Capability of *GoogleDataflowPlatform* node.

Next, developers must configure all the properties discussed in the previous section and save the final topology using Winery. Finally, developers can export the modeled Service template as a CSAR file.

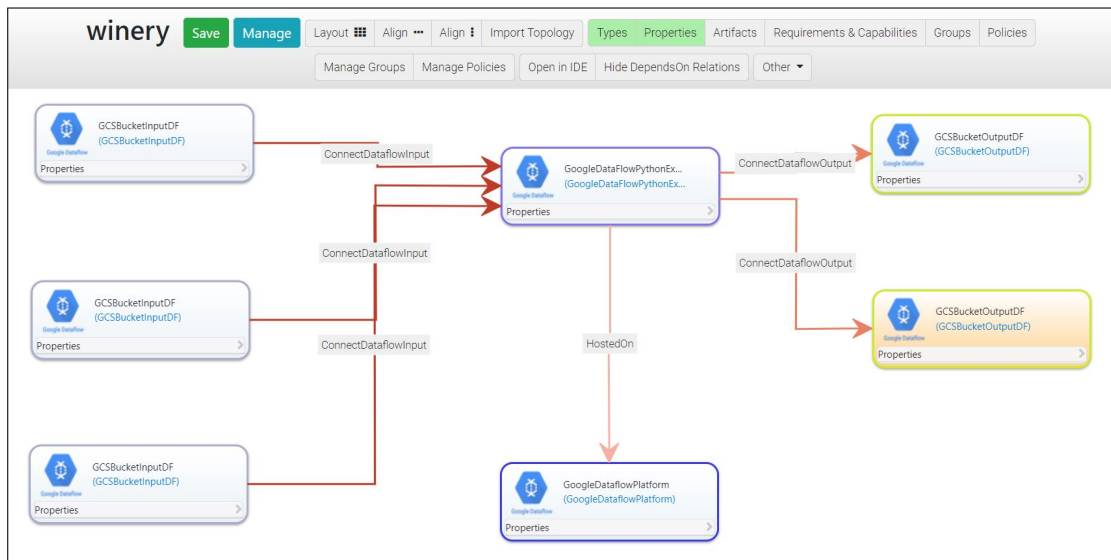


Figure 17. Designing Service Template *GoogleDataFlowPythonWordCount* in Winery

### 5.3.4 Deploying the exported CSAR file using xOpera

The exported CSAR file from the previous section is deployed using the below command.

```
opera deploy -c GoogleDataFlowPythonWordCount.csar
```

Figure 18 shows the Dataflow xOpera deployment logs for the TOSCA deployment. After the CSAR file deployment is complete, the Google Dataflow job execution should start.

Next, developers may stop the Dataflow job using xOpera. For that purpose, developers may create a new Dataflow job using the Winery. The new dataflow job may be designed in Winery with a similar steps as described in this example. However, developers must supply a different job name using the *dataflow\_job\_name* property of the *GoogleDataFlowPythonExecution* node type. After using the xOpera to deploy the new job, developers may wait for a few seconds for the job to appear on Google cloud console, and then undeploy the Dataflow job using below xOpera command:

```
opera undeploy
```

Figure 19 shows the xOpera logs for the undeploy method.

```

(..venv) ubuntu@my-xopera:~/dp-project$ opera deploy -c GoogleDataFlowPythonWordCount.csar
[Worker_0] Deploying GCSBucketOutputDF_0_0
[Worker_0]   Executing create on GCSBucketOutputDF_0_0
[Worker_0]   Executing configure on GCSBucketOutputDF_0_0
[Worker_0]   Executing start on GCSBucketOutputDF_0_0
[Worker_0] Deployment of GCSBucketOutputDF_0_0 complete
[Worker_0] Deploying GoogleDataflowPlatform_0_0
[Worker_0]   Executing create on GoogleDataflowPlatform_0_0
[Worker_0] Deployment of GoogleDataflowPlatform_0_0 complete
[Worker_0] Deploying GCSBucketOutputDF_1_0
[Worker_0]   Executing create on GCSBucketOutputDF_1_0
[Worker_0]   Executing configure on GCSBucketOutputDF_1_0
[Worker_0]   Executing start on GCSBucketOutputDF_1_0
[Worker_0] Deployment of GCSBucketOutputDF_1_0 complete
[Worker_0] Deploying GoogleDataFlowPythonExecution_0_0
[Worker_0]   Executing create on GoogleDataFlowPythonExecution_0_0
[Worker_0]   Executing pre_configure_source on GoogleDataFlowPythonExecution_0_0--GCSBucketOutputDF_0_0
[Worker_0]   Executing pre_configure_target on GCSBucketInputDF_2_0--GoogleDataFlowPythonExecution_0_0
[Worker_0]   Executing pre_configure_target on GCSBucketInputDF_0_0--GoogleDataFlowPythonExecution_0_0
[Worker_0]   Executing pre_configure_target on GCSBucketInputDF_1_0--GoogleDataFlowPythonExecution_0_0
[Worker_0]   Executing configure on GoogleDataFlowPythonExecution_0_0
[Worker_0]   Executing start on GoogleDataFlowPythonExecution_0_0
[Worker_0] Deployment of GoogleDataFlowPythonExecution_0_0 complete
[Worker_0] Deploying GCSBucketInputDF_2_0
[Worker_0]   Executing create on GCSBucketInputDF_2_0
[Worker_0]   Executing configure on GCSBucketInputDF_2_0
[Worker_0]   Executing start on GCSBucketInputDF_2_0
[Worker_0] Deployment of GCSBucketInputDF_2_0 complete
[Worker_0] Deploying GCSBucketInputDF_0_0
[Worker_0]   Executing create on GCSBucketInputDF_0_0
[Worker_0]   Executing configure on GCSBucketInputDF_0_0
[Worker_0]   Executing start on GCSBucketInputDF_0_0
[Worker_0] Deployment of GCSBucketInputDF_0_0 complete
[Worker_0] Deploying GCSBucketInputDF_1_0
[Worker_0]   Executing create on GCSBucketInputDF_1_0
[Worker_0]   Executing configure on GCSBucketInputDF_1_0
[Worker_0]   Executing start on GCSBucketInputDF_1_0
[Worker_0] Deployment of GCSBucketInputDF_1_0 complete

```

Figure 18. xOpera deployment logs for WordCount Dataflow

### 5.3.5 Evaluating final results

This example verifies the Dataflow job execution using the Google cloud console after the xOpera deployment in Section 5.3.4. Figure 23 from Google Cloud Console illustrates the Directed Acyclic Graph (DAG) for the WordCount Dataflow job. The Dataflow DAG in the figure demonstrates the sequence of steps performed for this Dataflow. The DAG starts with three parallel tasks, namely *Read1*, *Read2*, and *Read3*. The three input files loaded using these three parallel tasks are unified together using and *Flatten* task. The next three sequential tasks *Split*, *PairWithOne*, and *GroupAndSum* are group of transformations and processing applied to retrieve the word count. Next, *Format* task is used to format the output as per the requirement. Finally, the Dataflow writes the output to supplied GCS buckets using *Write1* and *Write2* tasks. One interesting point to note is that the Dataflow DAG in Figure 23 is generated based on the workflow instructions specified in the Python file from Listing 10.

As the next step, this example verifies that the Dataflow job saves the output word count file in the supplied output GCS buckets. For that purpose, GCS locations **gs://dp-**



```

(.venv) ubuntu@my-xopera:~/dp-project$ opera undeploy
[Worker_0] Undeploying GCSBucketInputDF_2_0
[Worker_0] Executing stop on GCSBucketInputDF_2_0
[Worker_0] Executing delete on GCSBucketInputDF_2_0
[Worker_0] Undeployment of GCSBucketInputDF_2_0 complete
[Worker_0] Undeploying GCSBucketInputDF_0_0
[Worker_0] Executing stop on GCSBucketInputDF_0_0
[Worker_0] Executing delete on GCSBucketInputDF_0_0
[Worker_0] Undeployment of GCSBucketInputDF_0_0 complete
[Worker_0] Undeploying GCSBucketInputDF_1_0
[Worker_0] Executing stop on GCSBucketInputDF_1_0
[Worker_0] Executing delete on GCSBucketInputDF_1_0
[Worker_0] Undeployment of GCSBucketInputDF_1_0 complete
[Worker_0] Undeploying GoogleDataFlowPythonExecution_0_0
[Worker_0] Executing stop on GoogleDataFlowPythonExecution_0_0
[Worker_0] Executing delete on GoogleDataFlowPythonExecution_0_0
[Worker_0] Undeployment of GoogleDataFlowPythonExecution_0_0 complete
[Worker_0] Undeploying GCSBucketOutputDF_0_0
[Worker_0] Executing stop on GCSBucketOutputDF_0_0
[Worker_0] Executing delete on GCSBucketOutputDF_0_0
[Worker_0] Undeployment of GCSBucketOutputDF_0_0 complete
[Worker_0] Undeploying GoogleDataflowPlatform_0_0
[Worker_0] Executing delete on GoogleDataflowPlatform_0_0
[Worker_0] Undeployment of GoogleDataflowPlatform_0_0 complete
[Worker_0] Undeploying GCSBucketOutputDF_1_0
[Worker_0] Executing stop on GCSBucketOutputDF_1_0
[Worker_0] Executing delete on GCSBucketOutputDF_1_0
[Worker_0] Undeployment of GCSBucketOutputDF_1_0 complete
(.venv) ubuntu@my-xopera:~/dp-project$

```

Figure 19. xOpera un-deploy logs for WordCount Dataflow

**output1/my\_output** and **gs://dp-output2/my\_output** are checked for the presence of output file using Google Cloud Console. Figure 21 and Figure 22 shows the output WordCount file in the both the specified GCS buckets. The output word count files are downloaded from the GCS buckets and included in the GitHub repository[19] for this thesis.

Finally, this example verifies the Dataflow job execution state in the Google cloud console after running the xOpera un-deployment in Section 5.2.4. Figure 23 shows the state of the Dataflow job after the same. As seen in the figure, each DAG step is shown in the Failed state, meaning the Dataflow job has failed and stopped executing. As expected after the opera un-deploy command, the WordCount job is canceled before completion, and no output is generated in the output GCS buckets.



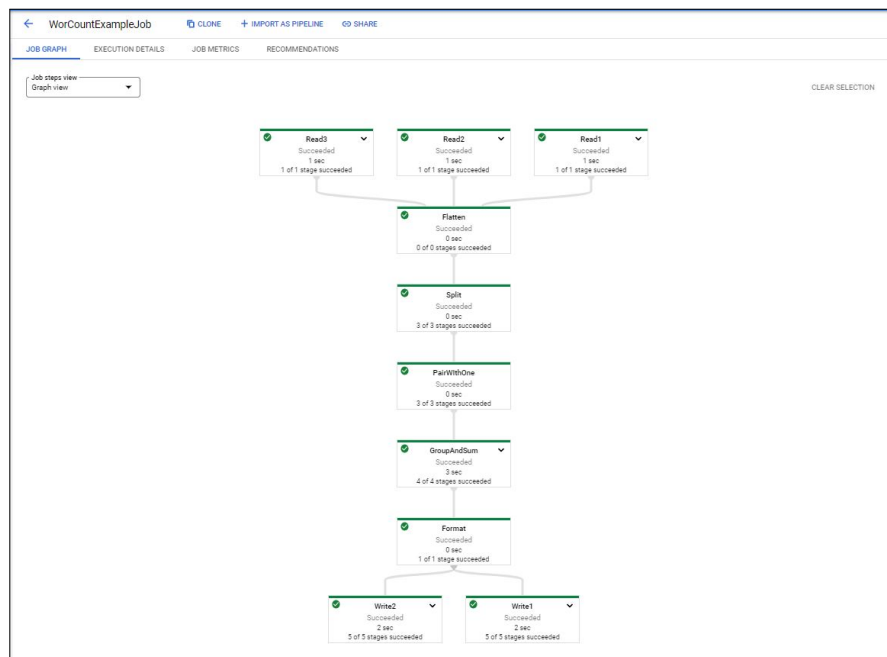


Figure 20. WordCount Job workflow in Google cloud console

dp-output1

Location

Storage class

Public access

Protection

europa-west1 (Belgium)

Standard

Not public

None

OBJECTS

CONFIGURATION

PERMISSIONS

PROTECTION

LIFECYCLE

Buckets > dp-output1 > my\_output

UPLOAD FILES

UPLOAD FOLDER

CREATE FOLDER

MANAGE HOLDS

DOWNLOAD

DELETE

Filter by name prefix only

Filter

Filter objects and folders

Show deleted data


<input type="checkbox"/>	Name	Size	Type	Created	Storage class	Last modified	Public access	Version history
<input type="checkbox"/>	 -00000-of-00001	88 KB	text/plain	25 Dec 2...	Standard	25 Dec 20...	Not public	—

Figure 21. Result of WordCount job stored in GCS Bucket dp-output1

dp-output2

Location

Storage class

Public access

Protection

europa-west1 (Belgium)

Standard

Not public

None

OBJECTS

CONFIGURATION

PERMISSIONS

PROTECTION

LIFECYCLE

Buckets > dp-output2

UPLOAD FILES

UPLOAD FOLDER

CREATE FOLDER

MANAGE HOLDS

DOWNLOAD

DELETE

Filter by name prefix only

Filter

Filter objects and folders

Show deleted data

<input type="checkbox"/>	Name	Size	Type	Created	Storage class	Last modified	Public access	Version history
<input type="checkbox"/>	my_output/	—	Folder	—	—	—	—	—

Figure 22. Result of WordCount job stored in GCS Bucket dp-output2

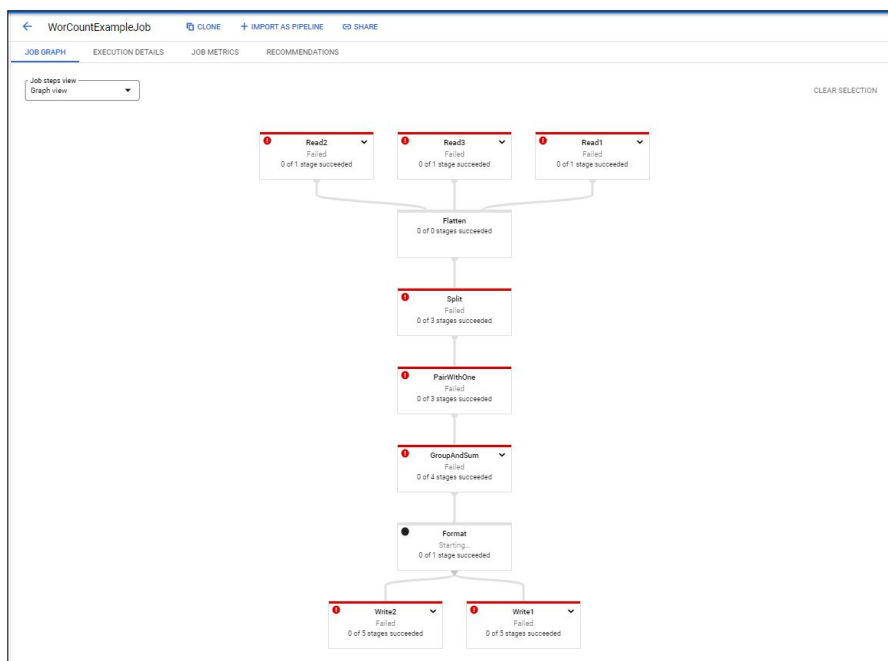


Figure 23. WordCount Job workflow in Google cloud console after job cancellation

## 5.4 Image Processing using Dataflow and Apache Nifi

This example orchestrates a data pipeline combining Google Dataflow and Apache Nifi. Figure 24 shows the high-level architecture for this image processing example. The data pipeline starts with Google Dataflow reading a image from a GCS bucket, resizing it, and publishing it to another GCS bucket. Then, Apache Nifi continues the data pipeline by fetching the resized image from the same GCS bucket and publishing it to the AWS S3 bucket. Like the previous example, the path to the python location is provided, and the Python code uses the Apache Beam library to define the Dataflow. The input and output locations are specified as runtime arguments to the Python code. Therefore, when the MidwayDF create operation generates the corresponding template from the Python code, then the generated template will also require the input and output arguments as parameters.

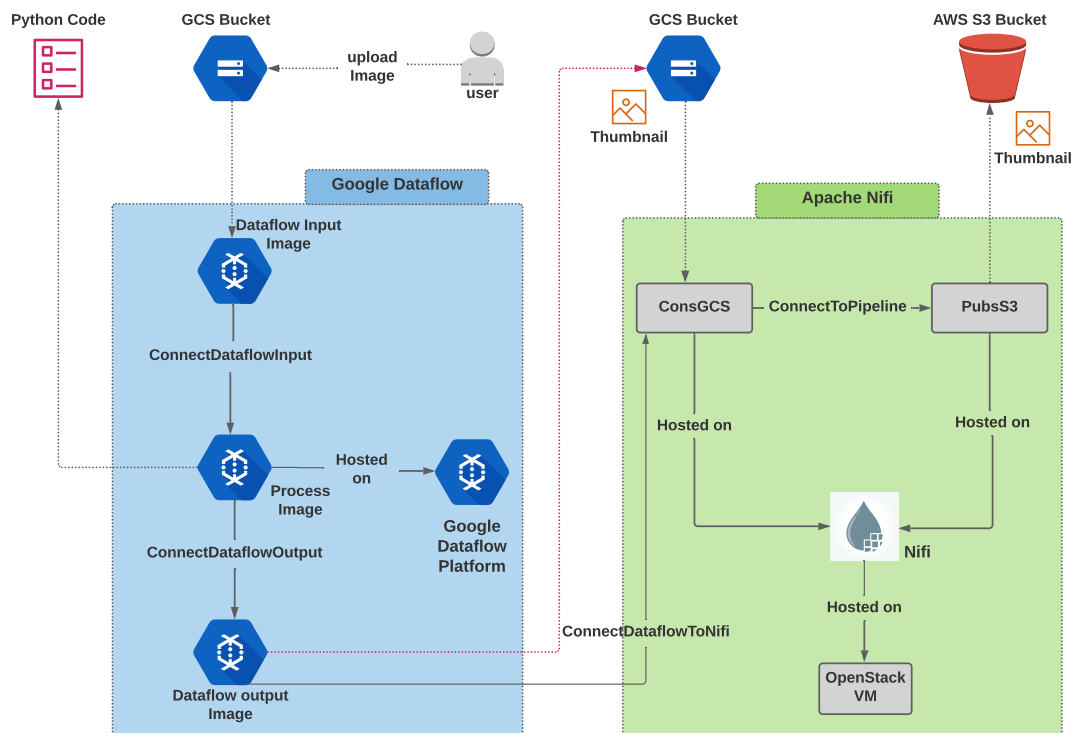


Figure 24. High-level architecture for image processing using Dataflow and Apache Nifi

### Input Image file

Input JPG image file is uploaded to the GCS bucket for resizing. Github repository [19] contains the image file that is uploaded to the source GCS bucket in this example. The

uploaded image file has dimension of 1280 X 720 pixels, which will be resized to 128 x 128 pixels using Dataflow to generate thumbnails.

#### 5.4.1 Prerequisites for this Dataflow

##### GCS buckets and uploading image files

The Google Dataflow reads the image from an existing GCS bucket, so this example creates a GCS bucket named **dp-input** and uploads a JPG image file. The input image file should be uploaded to the GCS location before deploying the Dataflow job. GCS bucket named **dp-output** is also created to store the Dataflow output image. Apache Nifi will use the same **dp-output** bucket to consume resized image and publish them to an AWS S3 bucket. This example also creates a GCS bucket **dp-bucket-2** to store the generated template and staging files for the Dataflow execution.

##### AWS S3 bucket

This example also creates an AWS S3 bucket to store the final resultant thumbnail image from this data pipeline. The Apache Nifi data pipeline consumes the processed images from the GCS bucket and publish it to the create AWS S3 bucket.

##### Python code using Apache Beam Library

Listing 11 shows the snippets from the written Python code for this image processing example. The class ImageThumbnailOptions uses the *add\_value\_provider\_argument* to declare the runtime parameters. developers must declare two runtime parameters for input and output GCS locations. The runtime parameters can then be supplied using GCSBucketInputDF and GCSBucketOutputDF node types.

---

```
class ImageThumbnailOptions(PipelineOptions):
    @classmethod
    def _add_argparse_args(cls, parser):
        parser.add_value_provider_argument('--inputBucket', help='
            Path of the file to read from')
        parser.add_value_provider_argument('--outputBucket', help='
            Output file to write results to.')

logging.getLogger().setLevel(logging.INFO)
pipeline_options = PipelineOptions()
pipeline_options.view_as(SetupOptions).save_main_session = True

with beam.Pipeline(options=pipeline_options) as p:
    def print_row(element):
        logging.info("Running pipeline:")

    def loadAndResize(path):
```

```

logging.info("Input:" + str(path))
buf = GcsIO().open(str(path), mime_type="image/jpeg")
img = Image.open(io.BytesIO(buf.read()))
resizedImg = img.resize((150, 150))
b = io.BytesIO()
resizedImg.save(b, format='JPEG')
return {"path": path, "image": b.getvalue()}

def store(item):
    output_path = my_options.outputBucket.get()
    name = item["path"].split("/")[-1].split(".")[0]
    path = os.path.join(output_path, f"{name}_thumbnail.jpg")
    writer = filesystems.FileSystems.create(path)
    writer.write(item['image'])
    writer.close()
my_options = pipeline_options.view_as(ImageThumbnailOptions)

(p
| "Start" >> beam.Create([""])
| "ReadInputParam" >> beam.Map(lambda x: my_options.inputBucket.
    get())
| "LoadAndResize" >> beam.Map(loadAndResize)
| 'Write' >> beam.ParDo(store))

p.run().wait_until_finish()

```

---

Listing 11. Python Code extracts for image processing example

### 5.4.2 Choosing the appropriate node types and supplying properties

This example uses Google Dataflow and Apache Nifi node types to design a combined data pipeline. For the Dataflow, developers must select *GCSBucketInputDF* as the *SourceDF* to receive image file from the GCS bucket. Since Python code is used to design the Dataflow, developers must use *GoogleDataFlowPythonExecution* as the *MidwayDF*. Next, this example uses *GCSBucketOutputDF* as the *DestinationDF* to specify the GCS bucket to publish the resulting thumbnail image. For the Apache Nifi data pipeline, this example selects *ConsGCSBucket* and *PubsS3Bucket* as the *SourcePB* and *DestinationPB*, respectively. The *ConsGCSBucket* will read the processed image file from the same bucket specified in *GCSBucketOutputDF*. The *ConsGCSBucket* then connects to *PubsS3Bucket* to publish the processed images to an AWS S3 Bucket. Nifi node type is selected to host the *ConsGCSBucket* and *PubsS3Bucket* node types. This Nifi node type is derived from *SoftwareComponent* node type, and its purpose is to install Apache Nifi Software. Finally, *Nifi* node type is hosted on *OpenStack* node type, that is used to instantiate an OpenStack VM. The selected node types along with their properties are discussed below.

### ***GCSBucketInputDF***

This node type receives the GCS location of the image file as input into the Google Dataflow. Table 10 discusses the mandatory properties and supplied value for the same.

Property	Value	Description
input	gs://dp-input/a3.jpg	This is the GCS Bucket location for reading the input image file.
param_name	inputBucket	This is the parameter name expected by the template for the 'input' property provided above. This is defined as an IO argument in the Python code.

Table 10. Properties for the *GCSBucketInputDF* node type in image processing Dataflow

### ***GoogleDataFlowPythonExecution***

This node type receives the absolute file path of the written Python code to create the Google Dataflow job. Table 11 showcases the properties and the corresponding supplied values for the GoogleDataFlowPythonExecution node type.

Property	Value	Description
dataflow_python_code_path	/home/ubuntu/dp-project/imageProcessing.py	This is the absolute path to the dataflow python file.
template_location	gs://dp-bucket-2/templates/my-template.json	The Dataflow template generated from the provided Python code is stored at this GCS location.
pypi_dependencies	["pillow"]	Supply required python dependencies to execute supplied Python code as a list. The Python code requires Pillow image processing library to resize images.
template_specific_parameters	None	No additional parameters need to be provided for this Dataflow job.
dataflow_job_name	ImageProcessingJob	The name of the Google Dataflow job.
project_id	thesis-project-329917	The Google project id where the Dataflow job will run.
credential_file_path	/home/ubuntu/dp-project/cred/thesis-project-329917-5c40808bbb3e.json	The system path to the Service Account JSON key. It provides authorization for the Dataflow and the configured project.
staging_location	gs://dp-bucket-2/staging	Google Cloud location where developers want to store the temporary and intermediate files.
region	europe-west1	This is the cloud region where developers want to execute the Dataflow job.

Table 11. Properties for *GoogleDataFlowPythonExecution* node type in image processing Dataflow

### ***GCSBucketOutputDF***

This node type receives the output GCS location for Dataflow to publish the processed image. Table 12 discusses the mandatory properties and supplied value for the same.

Property	Value	Description
output	gs://dp-output	This is the GCS Bucket location for publishing the processed image.
param_name	outputBucket	This is the parameter name expected by the template for the 'output' property provided above. This is defined as an IO argument in the Python code.

Table 12. Properties for the *GCSBucketOutputDF* node type in image processing Dataflow

### ***GoogleDataflowPlatform***

This node type hosts the *GoogleDataFlowTemplateExecution* node type. It does not require any mandatory property to be configured. This node type is responsible for installing the Google Cloud SDK and Apache Beam SDK that enables the host to communicate with Google Dataflow APIs.

### ***ConsGCSBucket***

This SourcePB node type configures the Apache Nifi processors under the hood to fetch objects from the GCS bucket. This node type integrates the Apache Nifi data pipeline with the created Dataflow. After Dataflow is done publishing the processed image to the GCS bucket, *ConsGCSBucket* fetches the same processed image as Flowfile. Table 13 describes the properties and supplied values for this node type.



Property	Value	Description
bucket	dp-output	This is the GCS Bucket location for reading the input image. For Apache Nifi nodes, users do not need to provide the protocol <b>gs://</b> .
project_ID	thesis-project-329917	The Google project id where the GCS bucket is located. This property matches the project_id property in the <i>GoogleDataFlowPythonExecution</i> node type.
credential_JSON_file	{"get_artifact": ["SELF", "credentialsGoogle"]}	The credential JSON file is supplied as an artifact to this node type. one should name the artifact as credentialsGoogle because developers expect the same file name using the get_artifact method as seen in provided value.
schedulingStrategy (optional)	EVENT_DRIVEN	Leave the value to default. This will trigger the Nifi data pipeline with every incoming image file.
name (optional)	consGCS-BucketNode	Name of the pipeline node.
scheduling-PeriodCRON (optional)	* * * * *	Leave the value to default.

Table 13. Properties for the *ConsGCSBucket* node type in image processing Dataflow

### ***PubsS3Bucket***

This DestinationPB node type receives the processed images files from the ConsGCS-Bucket node and publishes them to an AWS S3 bucket. Table 14 describes the essential properties and supplied value for this node type.

Property	Value	Description
BucketName	dp-aws-output-bucket	This is the AWS S3 Bucket location for publishing the processed image. For Apache Nifi nodes, users do not need to provide the protocol <b>s3://</b> .
Region	eu-west-1	The AWS region code for the bucket.
cred_file_path	{"get_artifact": ["SELF", "credentials"]}	The credential file is supplied as an artifact to this node type. one should name the artifact as credentials because same file name is expected by the get_artifact method as seen in provided value.
schedulingStrategy (optional)	EVENT_DRIVEN	Leave the value to default. This will trigger the Nifi data pipeline with every incoming image file.
name (optional)	PubsS3BucketNode	Name of the pipeline node.
scheduling-PeriodCRON (optional)	* * * * *	Leave the value to default.

Table 14. Properties for the *PubsS3Bucket* node type in image processing Dataflow

### ***Nifi***

This node type is responsible for installing the Apache Nifi software, and thereby, this node type hosts both *ConsGCSBucket* and *PubsS3Bucket* node types. Table 15 describes the mandatory properties and supplied values for this node type.

Property	Value	Description
port	8080	The port on which Apache Nifi is installed.
webinterface	true	Whether access to Apache Nifi web interface is allowed over public IP, this is set to true to access the web interface for verification purposes.
component_version	1.13.1	The Apache Nifi version that is to be installed.

Table 15. Properties for the *Nifi* node type in image processing Dataflow

### ***Workstation***

This node type denotes the physical server or VM using which RADON orchestrator deploys the Service Template. This node type uses the local server to host the Apache Nifi software, which will host the *ConsGCSBucket* and *PubsS3Bucket* node types. Table 16 discussed the properties and supplied values for this node type.

Property	Value	Description
pypi_dependencies	["jmespath"]	The Python dependencies to be installed on the local physical server or VM. This property accepts a list of python dependencies.

Table 16. Properties for the *Workstation* node type in image processing Dataflow

### 5.4.3 Designing the service template in the Winery

A service template named *ImageProcessingWithDataflowAndNifi* is created using Winery to model the Image processing data pipeline using Dataflow and Apache Nifi. Figure 25 shows the data pipeline modeling using Winery. Developers must connect the node types involved using their Requirements and Capabilities.

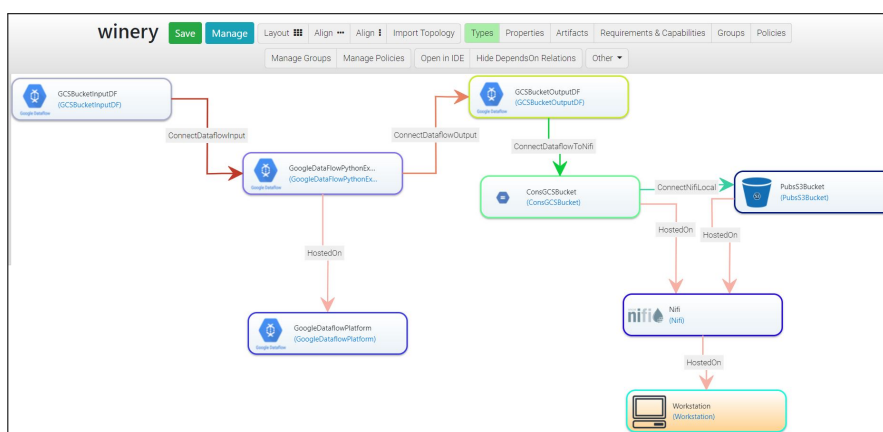


Figure 25. Designing the Service Template *ImageProcessingWithDataflowAndNifi* in Winery

The connection steps for Google Dataflow node types are described below:

- Dragging *ConnectDataflowInput* requirement from the *GCSBucketInputDF* node type into the *connectToDataflow* capability of *GoogleDataFlowPythonExecution* node type.
- Dragging *ConnectToDataflowOutput* Requirement from *GoogleDataFlowPythonExecution* into *connectToDataflow* Capability of the *GCSBucketOutputDF* node type.
- Dragging *HostedOn* Requirement from *GoogleDataFlowPythonExecution* into the *host* Capability of *GoogleDataflowPlatform* node type.

The connection steps for Apache Nifi data pipeline node types are described below:

- Dragging *ConnectNifiLocal* requirement from the *ConsGCSBucket* node type into the *ConnectToPipeline* capability of *PubsS3Bucket* node.
- Dragging *HostedOn* Requirement from *ConsGCSBucket* node type into the *host* Capability of *Nifi* node type.
- Dragging *HostedOn* Requirement from *PubsS3Bucket* node type into the *host* Capability of *Nifi* node type.
- Dragging *HostedOn* Requirement from *Nifi* into the *host* Capability of *Workstation* node type.

Finally, the connection step to integrate Google Dataflow node types with Apache Nifi node types is described below:

- Dragging *ConnectDataflowToNifi* requirement from the *GCSBucketOutputDF* node type into the *connectToDataflow* capability of *ConsGCSBucket* node type.

#### 5.4.4 Deploying the exported CSAR file using xOpera

The exported CSAR file from the previous section is deployed using the below command.

```
opera deploy -c ImageProcessingWithDataflowAndNifi.csar
```

Figure 26 shows the Dataflow xOpera deployment logs for the image processing data pipeline. After the CSAR file deployment is complete, the Google Dataflow job and Apache Nifi data pipeline execution should start.

Next, developers may stop the Dataflow job using xOpera using the below command:

```
opera undeploy
```

Figure 27 shows the xOpera logs for the un-deploy method.

#### 5.4.5 Evaluating final results

Dataflow job execution and Apache Nifi data pipeline deployment are verified after the xOpera deployment described in previous section. This paper validates the Dataflow and Apache Nifi node types integration by verifying the Dataflow job deployment, Apache Nifi data pipeline deployment, and the overall data processing and movement using the combined pipeline. Figure 28 shows the Cloud Console screenshot for Dataflow Dataflow DAG to understand the workflow. As specified in the Python file in Listing 11, The Dataflow DAG starts with the *Start* task, which is a Dataflow initializer. Next, the Dataflow uses *ReadInputParam* task to read the user input for the source GCS bucket. The Dataflow then loads and generates a thumbnail for the input image using the

```
(.venv) ubuntu@test2-vm:~/dp-project$ opera deploy -c ImageProcessingWithDataflowAndNifi.csar
[Worker_0] Deploying GoogleDataflowPlatform_0_0
[Worker_0] Executing create on GoogleDataflowPlatform_0_0
[Worker_0] Deployment of GoogleDataflowPlatform_0_0 complete
[Worker_0] Deploying Workstation_0_0
[Worker_0] Executing create on Workstation_0_0
[Worker_0] Deployment of Workstation_0_0 complete
[Worker_0] Deploying Nifi_0_0
[Worker_0] Executing create on Nifi_0_0
[Worker_0] Executing configure on Nifi_0_0
[Worker_0] Executing start on Nifi_0_0
[Worker_0] Deployment of Nifi_0_0 complete
[Worker_0] Deploying PubsS3Bucket_0_0
[Worker_0] Executing create on PubsS3Bucket_0_0
[Worker_0] Executing configure on PubsS3Bucket_0_0
[Worker_0] Executing start on PubsS3Bucket_0_0
[Worker_0] Deployment of PubsS3Bucket_0_0 complete
[Worker_0] Deploying ConsGCSBucket_0_0
[Worker_0] Executing create on ConsGCSBucket_0_0
[Worker_0] Executing configure on ConsGCSBucket_0_0
[Worker_0] Executing post_configure_source on ConsGCSBucket_0_0--PubsS3Bucket_0_0
[Worker_0] Executing start on ConsGCSBucket_0_0
[Worker_0] Deployment of ConsGCSBucket_0_0 complete
[Worker_0] Deploying GCSBucketOutputDF_0_0
[Worker_0] Executing create on GCSBucketOutputDF_0_0
[Worker_0] Executing configure on GCSBucketOutputDF_0_0
[Worker_0] Executing start on GCSBucketOutputDF_0_0
[Worker_0] Deployment of GCSBucketOutputDF_0_0 complete
[Worker_0] Deploying GoogleDataFlowPythonExecution_0_0
[Worker_0] Executing create on GoogleDataFlowPythonExecution_0_0
[Worker_0] Executing pre_configure_source on GoogleDataFlowPythonExecution_0_0--GCSBucketOutputDF_0_0
[Worker_0] Executing pre_configure_target on GCSBucketInputDF_0_0--GoogleDataFlowPythonExecution_0_0
[Worker_0] Executing configure on GoogleDataFlowPythonExecution_0_0
[Worker_0] Executing start on GoogleDataFlowPythonExecution_0_0
[Worker_0] Deployment of GoogleDataFlowPythonExecution_0_0 complete
[Worker_0] Deploying GCSBucketInputDF_0_0
[Worker_0] Executing create on GCSBucketInputDF_0_0
[Worker_0] Executing configure on GCSBucketInputDF_0_0
[Worker_0] Executing start on GCSBucketInputDF_0_0
[Worker_0] Deployment of GCSBucketInputDF_0_0 complete
```

Figure 26. xOpera deployment logs for Image Processing

```
(.venv) ubuntu@test2-vm:~/dp-project$ opera undeploy -r
The resume undeploy option might have unexpected consequences on the already undeployed blueprint.
Do you want to continue? (Y/n): Y
[Worker_0] Undeploying ConsGCSBucket_0_0
[Worker_0] Executing stop on ConsGCSBucket_0_0
[Worker_0] Executing delete on ConsGCSBucket_0_0
[Worker_0] Undeployment of ConsGCSBucket_0_0 complete
[Worker_0] Undeploying GoogleDataflowPlatform_0_0
[Worker_0] Executing delete on GoogleDataflowPlatform_0_0
[Worker_0] Undeployment of GoogleDataflowPlatform_0_0 complete
[Worker_0] Undeploying PubsS3Bucket_0_0
[Worker_0] Executing stop on PubsS3Bucket_0_0
[Worker_0] Executing delete on PubsS3Bucket_0_0
[Worker_0] Undeployment of PubsS3Bucket_0_0 complete
[Worker_0] Undeploying Nifi_0_0
[Worker_0] Executing stop on Nifi_0_0
[Worker_0] Executing delete on Nifi_0_0
[Worker_0] Undeployment of Nifi_0_0 complete
[Worker_0] Undeploying Workstation_0_0
[Worker_0] Undeployment of Workstation_0_0 complete
(.venv) ubuntu@test2-vm:~/dp-project$
```

Figure 27. xOpera un-deploy logs for Image Processing

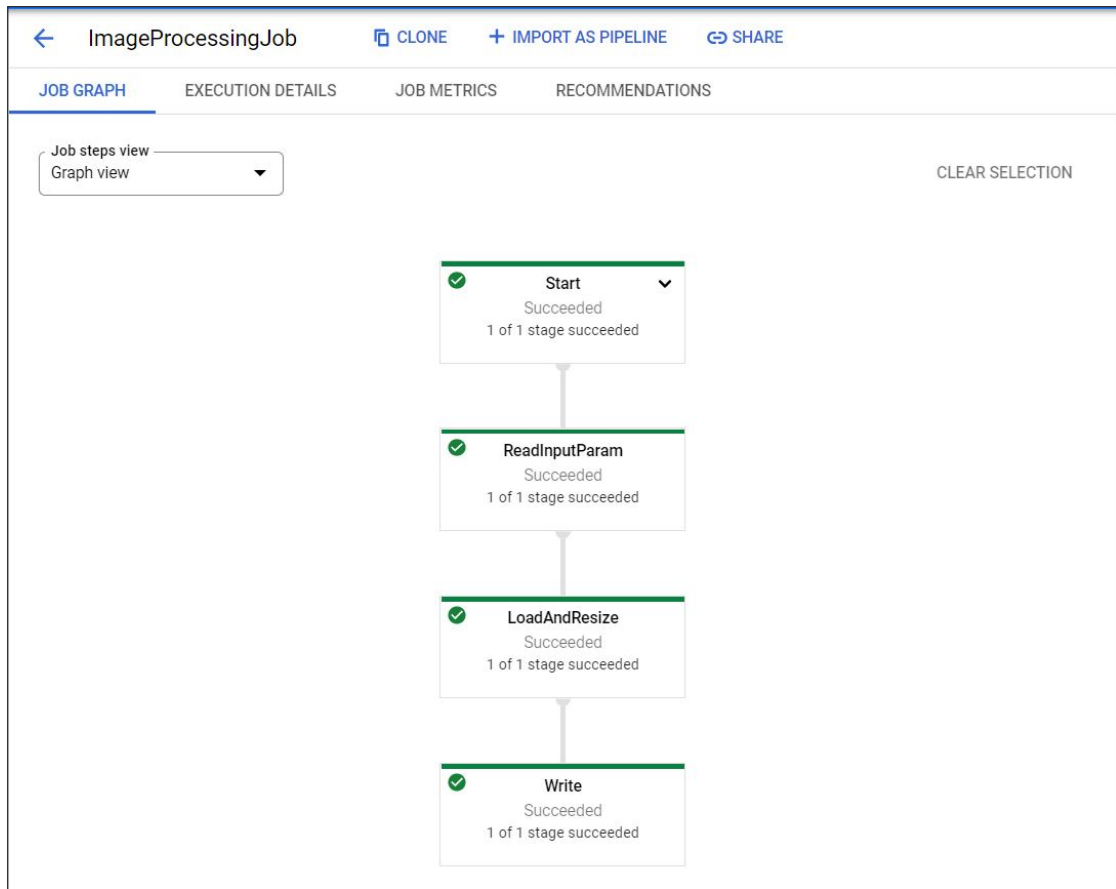


Figure 28. Image Processing Job workflow in Google cloud console

*LoadAndResize* task. Finally, Dataflow uses *Write* task to store the image thumbnail to the supplied output GCS bucket.

The Apache Nifi continues further with the data pipeline tasks. It picks up the output thumbnail image from the same GCS bucket where Google Dataflow has stored the output. The Apache Nifi then publishes the thumbnail image to an AWS S3 bucket. Figure 29 shows the Apache Nifi process groups in the running state. As seen in the figure, *GCSObject\_consume* process group passes a single Flowfile to the *S3Bucket\_dest\_PG\_LocalConn* process group. Figure 30 and Figure 31 show the screenshot of the GCS bucket and AWS bucket, respectively, with the resulting thumbnail image published. The GitHub repository[19] contains the supplied input and resulting thumbnail image.

For ease of demonstration, this example chooses to design Dataflow Python code for a single input image. However, developers can also design complex multi-file

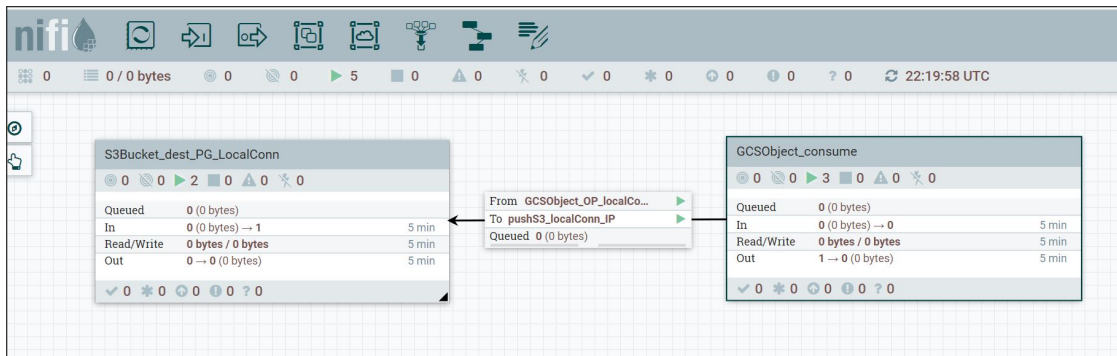


Figure 29. Running Process groups in Apache Nifi data pipeline

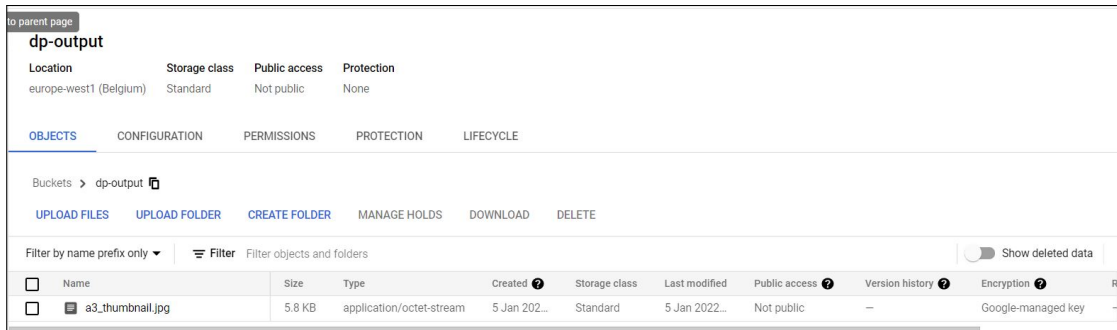


Figure 30. Output thumbnail image from Dataflow Image processing

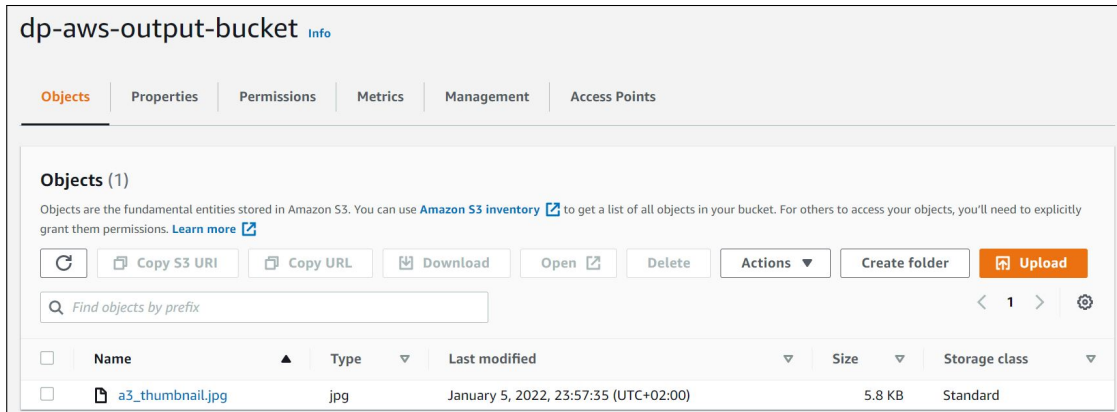


Figure 31. Output thumbnail image published to AWS S3 bucket by Apache Nifi

python projects following Apache Beam programming guidelines to design complex and

extensive Dataflow jobs. Finally, this example verifies the Dataflow job execution state in the Google cloud console after running the xOpera un-deployment in Section 5.2.4. The xOpera undeploy command uninstalls the Apache Nifi platform. Moreover, it also cancels the Dataflow job if it is in a running state, as demonstrated in previous examples.



## 6 Conclusion and future work

This section summarizes the thesis and also discusses future work. This thesis develops reusable and modular TOSCA components for the Google Dataflow in the RADON ecosystem, thus enabling developers to deploy and manage Google Dataflow without having platform-specific API knowledge. The use of the TOSCA standard to manage the Google Dataflow orchestration also enables developers to use their Google Dataflow in conjunction with other cloud services having pre-existing TOSCA implementation. The development work for this thesis enables developers to design extensive data pipelines in the RADON ecosystem combining Google Dataflow and Apache Nifi technology. The modular and extendible architecture for developed components allows room to implement additional new features for Google Dataflow and possible future integration with other pipeline technologies.

This thesis presents three sample data pipelines designed using the Service Templates with the help of developed TOSCA components. The Service Templates are then deployed using the proposed RADON orchestrator. The results are evaluated by verifying whether the Dataflow deployment and its possible integration with Apache Nifi are successful. This thesis evaluates the RADON orchestrator deployment logs, expected results in Google Cloud Console and Apache Nifi GUI to confirm whether the Dataflow deployment using developed TOSCA components were successful or not. The demonstrated examples showcase the multiple use-cases for the developed TOSCA node to create a batch and streaming Dataflow job, Template and Python-based Dataflow jobs, and Dataflow integration with Apache Nifi.

As for future work, there is still room for improvement. Firstly, the developed node types for the Google Dataflow could be extended to support scheduling features. The scheduling could be implemented using a Google Cloud scheduler or a cron job process running on a Google Compute Engine. Secondly, the TOSCA components could develop Google Dataflow jobs using Java programming language. This development will enable the users to write their Dataflow jobs using the Java programming language. Finally, the TOSCA components supporting Dataflow SQL could be designed that can take the SQL query to start the Dataflow jobs.

## References

- [1] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: issues and challenges. In *2010 24th IEEE international conference on advanced information networking and applications*, pages 27–33. Ieee, 2010.
- [2] Aiswarya Raj Munappy, Jan Bosch, and Helena Homström Olsson. Data pipeline management in practice: Challenges and opportunities. In *International Conference on Product-Focused Software Process Improvement*, pages 168–184. Springer, 2020.
- [3] OASIS. Tosca simple profile in yaml version 1.3. <https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.html>, 2020. Accessed: 2021-12-15.
- [4] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Tosca: portable automated deployment and management of cloud applications. In *Advanced Web Services*, pages 527–549. Springer, 2014.
- [5] Anže Luzar, Sašo Stanovnik, and Matija Cankar. Examination and comparison of tosca orchestration tools. In *European Conference on Software Architecture*, pages 247–259. Springer, 2020.
- [6] Antonio Brogi, José Carrasco, Javier Cubo, Francesco D’Andria, Elisabetta Di Nitto, Michele Guerriero, Diego Pérez, Ernesto Pimentel, and Jacopo Soldani. Seaclouds: an open reference architecture for multi-cloud governance. In *European Conference on Software Architecture*, pages 334–338. Springer, 2016.
- [7] Jacopo Soldani, Tobias Binz, Uwe Breitenbücher, Frank Leymann, and Antonio Brogi. Toscamart: a method for adapting and reusing cloud applications. *Journal of Systems and Software*, 113:395–406, 2016.
- [8] Chinmaya Dehury, Pelle Jakovits, Satish Narayana Srirama, Vasilis Tountopoulos, and Giorgos Giotis. Data pipeline architecture for serverless platform. In *European Conference on Software Architecture*, pages 241–246. Springer, 2020.
- [9] Chinmaya Kumar Dehury, Pelle Jakovits, Satish Narayana Srirama, Giorgos Giotis, and Gaurav Garg. Toscadata: Modeling data pipeline applications in tosca. *Journal of Systems and Software*, 186:111164, 2022.
- [10] Alex Brik and Jeffrey Xu. Diagnosing data pipeline failures using action languages: A progress report. In *International Symposium on Practical Aspects of Declarative Languages*, pages 73–81. Springer, 2020.

- [11] Apache nifi overview. <https://nifi.apache.org/docs.html>. Accessed: 2021-12-15.
- [12] Aws data pipeline. <https://docs.aws.amazon.com/datapipeline/latest/DeveloperGuide/what-is-datapipeline.html>. Accessed: 2021-12-15.
- [13] Cloud composer documentation. <https://cloud.google.com/composer/docs/concepts/overview>. Accessed: 2021-12-15.
- [14] Apache airflow documentation. <https://airflow.apache.org/docs/apache-airflow/stable/index.html>. Accessed: 2021-12-15.
- [15] Dataflow documentation. <https://cloud.google.com/dataflow/docs>. Accessed: 2021-12-15.
- [16] Giuliano Casale, Matej Artac, W-J van den Heuvel, André van Hoorn, Pelle Jakovits, Frank Leymann, Mike Long, Vasilis Papanikolaou, Domenico Presenza, Alessandra Russo, et al. Radon: Rational decomposition and orchestration for serverless computing. *SICS Software-Intensive Cyber-Physical Systems*, 35(1):77–87, 2020.
- [17] xopera. <https://xlab-si.github.io/xopera-docs/>. Accessed: 2021-12-15.
- [18] Chinmaya Dehury. Data pipeline orchestration ii. <https://radon-h2020.eu/wp-content/uploads/2021/09/D5.6-Data-pipeline-orchestration-II.pdf>, 2019. Accessed: 2021-12-17.
- [19] Thesis documents. <https://github.com/manishgupta94/thesis-documents>. Accessed: 2022-01-04.
- [20] Github repository for wordcount example. [https://github.com/apache/beam/blob/master/sdks/python/apache\\_beam/examples/wordcount.py](https://github.com/apache/beam/blob/master/sdks/python/apache_beam/examples/wordcount.py). Accessed: 2021-12-23.
- [21] Apache beam programming guide. <https://beam.apache.org/documentation/programming-guide/>. Accessed: 2021-12-17.

# **Appendix**

## **I. Abbreviation**

TOSCA - Topology and Orchestration Specification for Cloud Applications

GCP - Google Cloud Platform

GCS - Google Cloud Storage

SME - Small and Medium Enterprises

ETL - Extract, transform, load

IaaS - Infrastructure as a Service

PaaS - Platform as a Service

SaaS - Software as a Service

VM - Virtual Machine

DAG - Directed Acyclic Graph

API - Application Programming Interface

ETL - Extract, Transform, Load

IoT - Internet of Things

FTP - File Transfer Protocol

AWS - Amazon Web Services

EC2 - Elastic Compute Cloud

EMR - Elastic Map Reduce

Cloud ML - Cloud Machine Learning

CSAR - Cloud Service Archive

FaaS - Function as a Service

IO arguments - Input Output arguments

JSON - JavaScript Object Notation

CSV - Comma Separated Values

SDK - Software Development Kit

## II. Repositories

This Appendix contains link to important GitHub Repositories.

1. Fork of RADON particle GitHub Repository - This repository contains the developed TOSCA components to orchestrate Google Dataflow. The repository can be accessed from <https://github.com/manishgupta94/radon-particles>
2. Used images, files and screenshots repository - This repository contains used figures, used input and output files to validate the developed node types, used screenshots for this thesis. The repository has the all the files classified based on the sections. This repository can be accessed from <https://github.com/manishgupta94/thesis-documents>

### III. Technical Manual

This appendix describes the complete technical manual for orchestrating Google Dataflow using RADON tools like Winery and xOpera. First, we set up the RADON graphical modeling tool Winery. Next, we look at the steps of designing a service template in Winery for Google Dataflow presented in Section 5.2. In the next step, we set up the RADON orchestrator xOpera, and finally, we look at the steps of deploying the designed service template using xOpera. For the mentioned steps, we need a VM or physical server with Operating System installed. This technical manual will demonstrate these steps for a VM server with Ubuntu 18.04 installed.

#### Setting up Winery

The first step toward setting up Winery is to install the docker. The installation steps are described below.

- Create a directory in the Linux host in the directory /etc/docker using the command:

```
sudo mkdir /etc/docker
```

- Create a file daemon.json in the directory /etc/docker using the command:

```
sudo vi /etc/docker/daemon.json with below content.
```

---

```
{
  "default-address-pools": [{ "base": "172.80.0.0/16", "
                             size": 24 }]
}
```

---

- Update the package repo: `sudo apt-get update`

- Install dependency packages for docker.

```
sudo apt-get install apt-transport-https ca-certificates curl gnupg-agent
software-properties-common
```

- Add Docker's GPG key:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

- Set up the stable repository using:

```
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu
$(lsb_release -cs) stable"
```

- Update the repository and install docker.

```
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Create a remote github repository using the source code provided with this thesis. Let us assume for this demonstration that the link to the remote repository is:

<https://github.com/manishgupta94/radon-particles> Run the below command to run a docker container with the Winery image from the Docker Hub.

```
docker run -itd -p 8080: 8080 \  
-e PUBLIC_HOSTNAME = localhost \  
-e WINERY_FEATURE_RADON = true \  
-e WINERY_REPOSITORY_PROVIDER = yaml \  
-e WINERY_REPOSITORY_URL = https://github.com/manishgupta94/radon-particles \  
opentosca / radon-gmt
```

After the docker container for Winery is active, you can access Winery using the below url:

```
http://HOST_IP:8080
```

The HOST\_IP is the IP address of the host where the Winery is installed.

## Designing the Service Template in Winery

Create a Service Template in the Winery.

- Click on Service Template on the top left and click on the Add new button on the top right and a pop-up should appear.
- Edit the name for your Service Template, disable the versioning, and select the appropriate Namespace for your service template as shown in Figure 32.
- Click on Add to create your Service Template.

Open the Service Template in the Editor by clicking on Topology Template and then Open Editor button.

On the left Palette on the Canvas, drag and drop the following node types similar to example from section 5.2.

- **GCSBucketInputDF** - Drag three of these node types from the package  
`radon.nodes.datapipeline.source`
- **GoogleDataFlowPythonExecution** - Drag one of this node type from  
`radon.nodes.datapipeline.process`
- **GCSBucketOutputDF** - Drag two of this node type from  
`radon.nodes.datapipeline.destination`
- **GoogleDataflowPlatform** - Drag one of this node type from  
`radon.nodes.google`

Add a new Service Template

Name  
GoogleDataFlowPythonWordCount

Versioning:  
☐ Enable Versioning

Namespace  
radon.blueprints.examples

Template: ▼

Cancel Add

Figure 32. Creating a Service Template

Configure the properties for all the nodetypes by clicking on the properties button on top of the canvas. The value of the properties can be fetched from the section 5.2.

Configure the Requirements and Capabilities by clicking on the Requirements and Capabilities button on the top of the canvas. Follow the below steps to connect the node types. The end result should look like as shown in Figure 13.

- Drag *ConnectDataflowInput* Requirement from all three *GCSBucketInputDF* node types into *connectToDataflow* Capability of *GoogleDataFlowPythonExecution* node.
- Drag *ConnectToDataflowOutput* Requirement from *GoogleDataFlowPythonExecution* into *connectToDataflow* Capability of both the *GCSBucketOutputDF* node types.
- Drag *HostedOn* Requirement from *GoogleDataFlowPythonExecution* into *host* Capability of *GoogleDataflowPlatform* node.

In case, any file needs to be provided as an artifact, then click on Artifacts button on the top of the canvas. Expand Artifact for *GoogleDataFlowPythonExecution*, and click on add artifact to add the required artifact.



After you are done designing your service template, click on save button on the top left. Click on manage next and then Click on Export > Download to export the Service template as CSAR file.

### Setting up RADON xOpera

Install the RADON xOpera by following the below steps.

- Connect to the Linux host and update the repository `sudo apt-get update`
- Install the Python virtual environment  
`sudo apt install -y python3-venv python3-wheel python-wheel-common`
- Create a new dp-project directory `sudo mkdir /dp-project`
- Move into the newly created directory  
`cd /dp-project`
- Create a Python virtual env  
`python3 -m venv .venv`
- Activate the virtual environment.  
`source .venv/bin/activate`
- Update the pip.  
`pip install --upgrade pip`
- Install opera inside the virtual environment.  
`pip install opera`

Now opera is installed inside the virtual environment. Make sure you are inside the virtual environment before you use opera. If you see a `(.venv)` in the beginning of the command line prompt, that means you are inside the virtual environment. If not, then follow below steps to gain access.

- `cd /dp-project`
- `source .venv/bin/activate`

### Deploying and Undeploying a service template

Copy the exported CSAR file to the Linux host where you installed xOpera and make sure you are inside Python virtual environment. Run the below command to deploy your CSAR file.

```
opera deploy -r GoogleDataFlowPythonWordCount.csar
```

Once the deployment is successful, you can go and verify the results in the Google cloud console.

Similarly, run the below command to undeploy our CSAR file.

```
opera undeploy
```

## IV. Licence

### Non-exclusive licence to reproduce thesis and make thesis public

I, **Manish Gupta**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

**Google Dataflow orchestration using TOSCA in the hybrid cloud,**

supervised by Chinmaya Dehury and Pelle Jakovits.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.
4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Manish Gupta

**06/01/2022**